Humphries

ISSN 0265-2919

90p 77

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO

An ORBIS Publication

IR£1.15 Aus $2.15 NZ $2.65 SA R2.45 Sing $4.50

CONTENTS

APPLICATION

DRAWING ROOM Computer aided design is now widely used in industry. We examine this powerful tool — 1521

HARDWARE

HOT OFF THE PRESS Apple's new LaserWriter printer has brought typeset quality text to the microcomputer — 1529

SOFTWARE

DRESS REHEARSAL Having fleshed-out the characters in our drama, we now move on to handling their individual attributes — 1526

COMPUTER SCIENCE

RATIONAL SAVINGS Our tutorial on FORTH now considers the use of colon definitions and methods of compiling — 1535

JARGON

FROM RS232C TO SAWTOOTH A weekly glossary of computing terms — 1528

PROGRAMMING PROJECTS

OBLIQUE STRATEGIES We give routines to deal with stone groupings for the Amstrad, Commodore 64 and Spectrum — 1532

MACHINE CODE

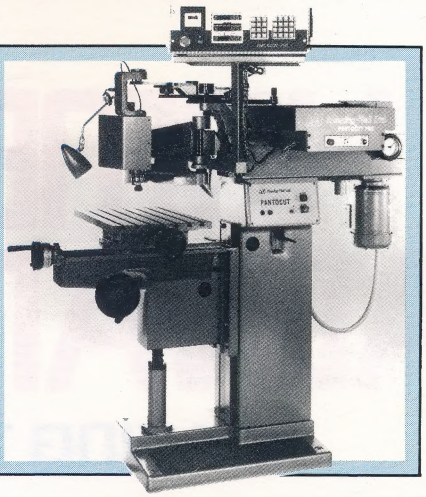
AMSTRAD ON LOCATION We begin a short series on the Amstrad CPC 464/664 with an overview of the operating system — 1538

WORKSHOP

PINPOINTING THE SIGNALS We conclude our examination of the individual pins on the 7135 chip — 1524

Next Week

- **We conclude the Go series with an examination of some of the techniques which could be used to improve the program.**
- **The Amstrad operating series continues with a look at how the firmware on the CPC 464 and 664 handles input/output.**
- **Computers in industry examines the methods of numerical and process control which are used to control manufacturing machinery.**



QUIZ

1) Apart from the equipment provided by Robocom, what other equipment is required to run the Bitstik system on a BBC micro?
2) The Macintosh can transmit the information required to produce a page via the LaserWriter in as little as 8 Kbytes. Why then does the LaserWriter require 1.5 Megabytes of RAM on board?
3) The Amstrad BASIC command POKE &DDDD, PEEK(&DDDD) would appear to do nothing. What effect does this line actually produce?

Answers To Last Week's Quiz

1) The device which is the optical equivalent of the transistor is called the transphasor.
2) The display file tells the computer whether a pixel is on or off, whereas the attributes file gives information about the foreground and background colours.
3) AMT stands for advanced manufacturing techniques.

Coming Up

- A look at GEM, Digital Research's operating environment for 16-bit machines.
- Programming languages yet to be covered in Computer Science include FORTRAN and BCPL.

A BLASTED GOOD TIME The first rock musical written for computer — INSIDE BACK COVER

Editor Stephen Cooke; **Art Editor** Claudia Zeff; **Deputy Editor** Steve Colwill; **Production Editor** Bobby Pickering; **Designer** Julian Dorr; **Staff Writer** Steve Malone; **Art Assistant** Caroline Clayton; **Sub Editors** Jon Kaye, Nik Lumsden; **Contributors** Chris Honey, Chris Laing, Steve Cooke, Helena Sturridge, Marcus Jeffery, Steve Colwill, Steve Malone, Steven Vickers, David Mudd; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Assistant** Susan Brown; **Subscription Manager** Christine Allen; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1985; © Orbis Publishing Ltd 1985: Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Heanor Gate Printing Ltd, Derby**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders.
UK/EIRE – Issue Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Issue Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Issue Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Issue Price: US/CAN$1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Issue Price: SA R2.45. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Issue Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Issue Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Issue Price: Aus$2.15. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Issue Price: NZ$2.65. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 5211. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder price include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited. Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER ILLUSTRATION BY KEVIN JONES WITH APOLOGIES TO APPLE

DRAWING ROOM

Computer aided design (CAD) is now used extensively in industry, not solely as a means of producing detailed drawings, but also for testing products before they are made. We take a look at some of the techniques designers now have at their disposal.

The important difference that separates elementary computer graphics from true CAD is that the latter is concerned with more than just drawing the object. The most sophisticated CAD systems have a real sense of the object housed in their memories rather than a mere picture. Some systems are able to test the object, fitting working parts together, simulating stresses and strains and ensuring that the final product won't fall apart once it is made.

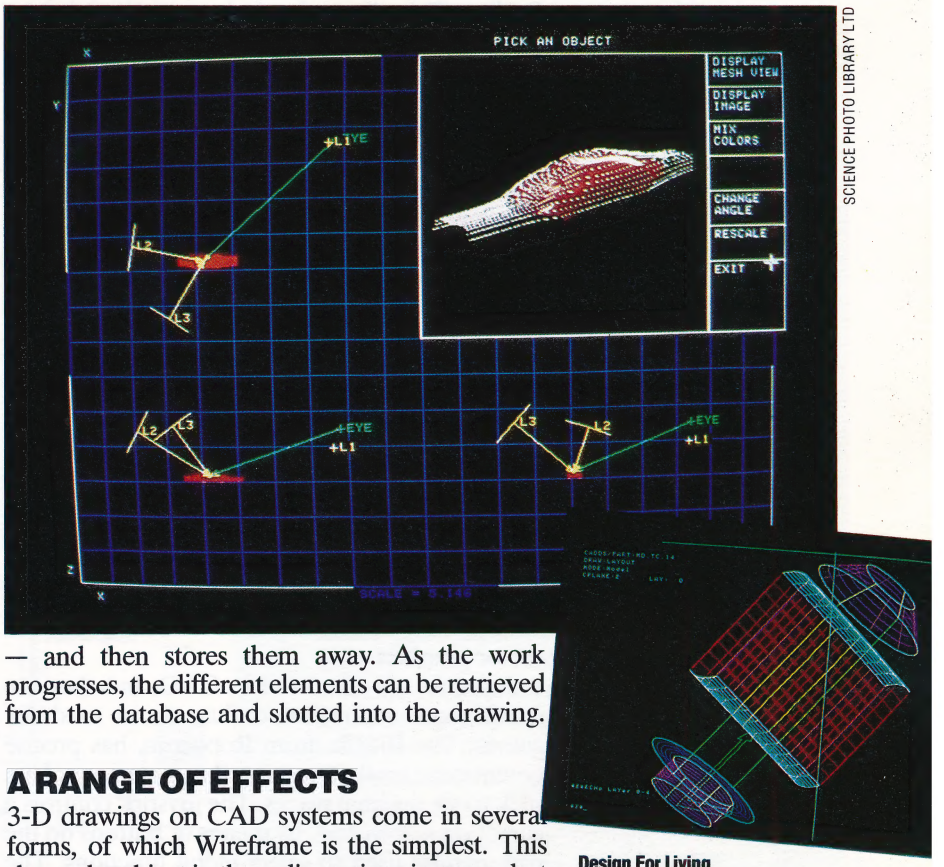
The pictures in most CAD systems are, in fact, created in a different way to those in ordinary graphics systems. Normal computer graphics build up images in pixels, but these are not precise enough for CAD, which uses instead the vector graphics system. This bases its drawing on the mathematical description of shapes. Thus, in pixel grahics, a line would be a row of dots on the screen, but in vector grahics it would be understood as the co-ordinates of either end and the shortest distance between them. Vector graphics are more accurate than pixel graphics, allowing the processor to deal with much higher resolutions than the monitor can display.

The simplest CAD systems draw in two dimensions and can sometimes add the appearance of a third — what is usually called $2\frac{1}{2}$-D drawing. Such a system can handle all the needs of an architect's office, where the extra 'half' dimension would allow the architect to show an elevation as well as a ground plan.

Companies using CAD to design electronic circuitry also use 2-D systems, but of greater complexity. Because CAD systems can hold information not just about a drawing but about the actual product, CAD in the field of electronics is usually coupled with test programs — once a chip designer has mapped out the millions of circuits on a microprocessor, the system will check to see that the circuitry functions correctly.

The most complex use of CAD is in designing objects for manufacture. It requires 3-D and therefore needs more memory and faster processing in order to manage the image. Since the designs are to be turned into manufactured products, it must be exceptionally accurate.

The designer using a CAD system first maps out the individual parts — each to a specific scale — and then stores them away. As the work progresses, the different elements can be retrieved from the database and slotted into the drawing.



SCIENCE PHOTO LIBRARY LTD

Design For Living
CAD uses vector graphics technology to manipulate images and display them in very high resolution, as shown in this screenshot of a weld assembly designed using CAD techniques. Advanced systems, however, will also enable designs to be tested prior to manufacture. In the second image,a Cray supercomputer at General Motors tests the aerodynamics of a new vehicle

A RANGE OF EFFECTS

3-D drawings on CAD systems come in several forms, of which Wireframe is the simplest. This shows the object in three dimensions in space, but only in outline. There is no image of the surfaces. Full 3-D, and solids modelling, provides the surfaces, and the system itself will have a sense of the object as it really is.

The most complex systems can produce coloured images, with up to 256 different colours on screen at the same time. Shading will add depth and help provide a more realistic image.

CAD users have a range of effects at their disposal when building up their pictures, beyond the ability to manipulate geometrical shapes. Some of the more important are:

Dimensioning: Once a drawing is completed the designer can have the system fill in all the measurements automatically, ready for drawing. Because the system holds information in numeric form this is a straightforward operation.

Hatching: Designers need to be able to highlight sections. Hatching is an automatic shading technique used by technical artists.

Layering: Designs can be built up in layers. An architect might keep the ground plan of a house on one layer and the drains and piping on another. Both could appear on the same printout or they could appear separately, depending on what information was needed.

Segments: These are sometimes called entities and are the individual elements that make up a drawing. They are held as a sort of catalogue of ready-made drawings that the designer can call up.

Rubber-Banding: This technique lets the designer push or pull lines about — like elastic — to place them where he needs them.

Scaling: This allows the designer to draw in any size he pleases, and then store the information to scale with the rest of the drawing.

Zoom and Pan: the first of these two cinematic terms means the user can focus on a section of a drawing and blow it up to fill the screen. Pan allows him to stand back, as it were, and view the whole drawing.

INPUT DEVICES

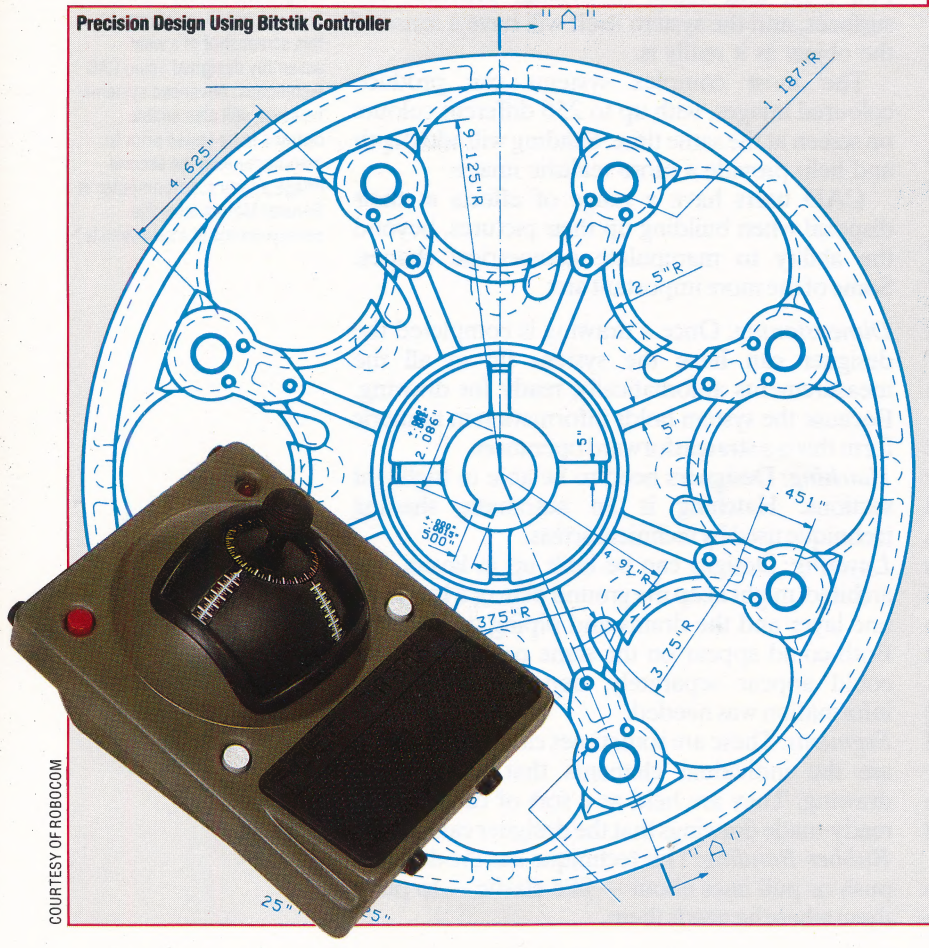
CAD users have a choice of devices they can use, depending on the type of work they are doing. Among the most familiar are the digitising pad, the light pen, the tracker ball and the mouse.

Image processors are used to input extremely complex designs and photographs. They work by photographing the image to be entered and breaking the photograph down into the shaded elements that make up a screen picture.

The keyboard is frequently used for inputting information on CAD systems, since it is a highly accurate way of entering the co-ordinates used in vector graphics.

Joysticks are also used in CAD, but they are much more sophisticated than those used for games. The Bitstik, from Robocom, has precise potentiometers that measure the movement of the stick to six decimal places. The joystick controls a cursor on screen and, by means of buttons on the pad, can be used to select menu functions. The end of the Bitstik's arm can be adjusted to zoom in on a detail or pan back to the whole picture.

Precision Design Using Bitstik Controller



COURTESY OF ROBOCOM

CAD On The Home Micro

Computer aided design (CAD) is becoming increasingly popular among designers as a reliable and cost-effective tool in helping to produce either original designs, or to test possible designs without the necessity of building expensive models. Although computer aided design requires a large quantity of equipment and peripherals to obtain professional results, many of the systems currently available are based around quite modest home micros.

The HRX system available for the Memotech 512 computer is a package which enables screens obtained from a video camera to be enhanced by the computer. 'Frames' from the video camera can be read into the computer system and a number of effects can be performed. Among these are the ability to zoom, shrink or rotate the picture. It is also possible to completely change the colours used in the picture. The colour system used by the HRX format uses the three basic colours of a video system (red, green and blue) and is capable of over 16.5 million colours per pixel. A normal television set only requires a 250,000 colour capability.

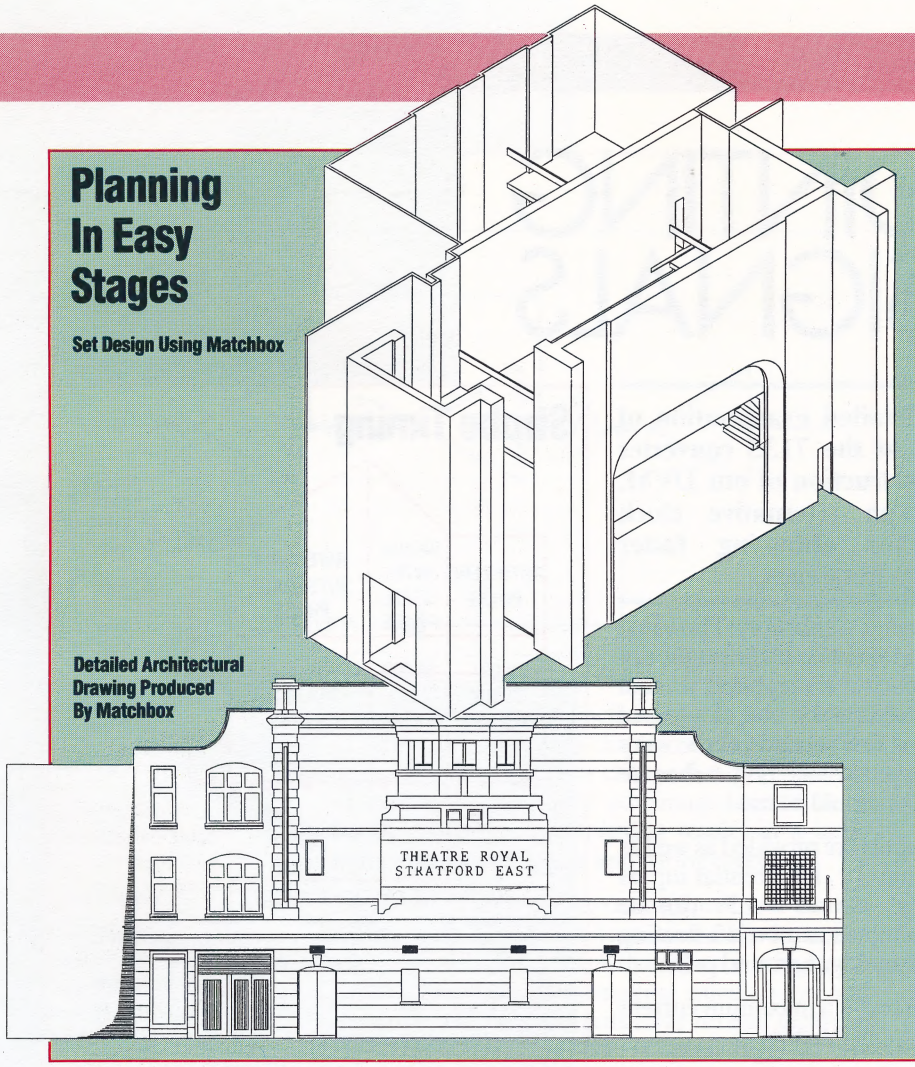
Of course, such a system requires a lot of additional hardware in order to function effectively. With so much information needing to be processed obviously a great deal of memory is required. The user, therefore, requires a double-width controller and memory card, with additional memory cards if colour is to be added. Furthermore, a special single (for monochrome) or three channel (colour) 'flash' A/D converter is required to accept information from the MTX computer. A fully equipped system will store data from a video camera at the rate of 7.2 Mbytes per second. Obviously, a system offering such power is quite costly and a complete HRX system will cost you several thousand pounds.

More down to earth is the Bitstik system for the BBC micro. This package allows the BBC micro to produce technical drawings of professional quality. The £795 system consists of a Bitstik controller — a kind of high precision joystick — a systems disk, an eight Kbyte applications ROM and a library disk containing a number of common typefaces and symbols. However, in order to run the system, the BBC micro also has to be fitted with a 6502 second processor and twin 80-track disk drives. In order to get the most from the system the user will also need a colour monitor and a colour plotter. The cost of a complete system will therefore be around £2,000.

However, for this the user gets a system which, the manufacturers say, will allow someone with no artistic ability to produce professional standard graphics designs. The basis of the system is the library of shapes held on disk, and the 'primitives', such as lines and arcs, held on ROM. Diagrams are constructed by building more complex designs from these primitives which can then be incorporated in larger designs. The central part of the system is the Bitstik controller itself, which allows the primitives to be positioned with pinpoint accuracy

Planning In Easy Stages

Set Design Using Matchbox



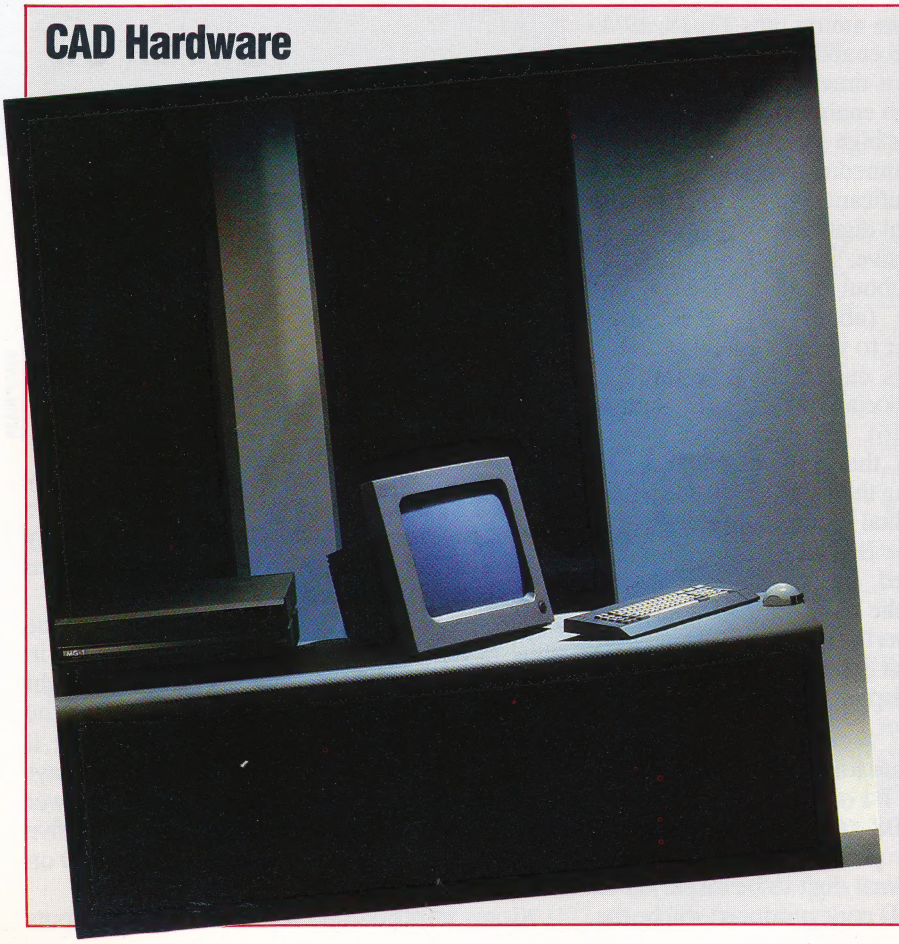
Detailed Architectural Drawing Produced By Matchbox

Although computer aided design is usually associated with the development of industrial products like cars, the techniques involved can be adapted to almost any application where 'modelling' is required. One of the uses which has been made of CAD in a non-industrial environment is in the theatre. A major headache for a set designer is to arrange the scenery in such a way as to produce the required visual effect, without obscuring the audience's view. The problem is compounded for touring productions which travel from theatre to theatre, each of which will have its own 'sight lines'.

Traditionally, this problem has been solved by the creation of models of the theatre with scaled reproductions of the sets which are moved around the stage until the optimum arrangement is found. However, a team of theatrical designers at the Theatre Royal, Stratford, in the East End of London have collaborated with design consultants Robary Ltd to produce Matchbox, a package which enables sets to be created on a VDU and viewed from any angle within any particular theatre. The advantages of the system are that a designer can create a set and take it through all the views and theatres in which it will be used. If the set does not suit a particular theatre, it can be changed, and props moved around the stage instantly, rather than having to rebuild the models from scratch. Furthermore, the Matchbox system will produce accurate plans and drawings from which the designers and builders can work

MODELBOX AT THE THEATRE ROYAL, STRATFORD EAST

CAD Hardware



It is not just software which is tailored for use in CAD applications. A measure of the importance seen in computer aided design, is that some computers are also being developed which give particular prominence to the needs of CAD. The MG-1 Workstation from Whitechapel Computer Works, is a minicomputer which contains a number of features essential to efficient CAD processing.

The MG-1 is based around the National Semiconductor NS32016 32-bit processor which provides the kind of fast processing power required in computer aided design. The graphics display has a resolution of 1024 × 800 pixels and, in order to produce fast flicker-free images, a number of advanced features are included. The machine is equipped with direct memory refresh, thus obviating the need for frame buffers which reduce the speed at which the VDU screen is refreshed. Furthermore, in order to reduce the problem of jerking cursor movement from the mouse, there is a separate processor to handle the mouse and keyboard input.

The MG-1 runs under GENIX. This is the National Semiconductor version of the popular Unix multitasking operating system. A number of packages have been developed to run on the MG-1 under this system. For example, the Nottingham-based firm Payfec has produced a professional draughtsman system, known as DOGS, while Lattice Logic have developed a processor design package known as the Chipsmith

COURTESY OF WHITECHAPEL COMPUTER CENTRE

PINPOINTING THE SIGNALS

We conclude our detailed examination of the individual pins of the 7135 converter chip used in the construction of our DVM, and also include an alternative clock oscillator circuit for obtaining faster measurements at high frequency.

7/8. REF. CAP. (Reference Capacitor): The value and quality of this component is, surprisingly, not very critical. A 1µF capacitor is suggested, though a smaller one may be used (at the cost of a loss of accuracy during the first few seconds of recovery after an overload). Polypropylene, polycarbonate or polystyrene capacitors should be used.

9/10 IN–/IN+: Two inputs are provided as well as an analogue ground (pin 3). Differential inputs anywhere in the range –4v to +4.5v can be accommodated. These inputs can be left floating or IN- can be tied to the analogue ground point.

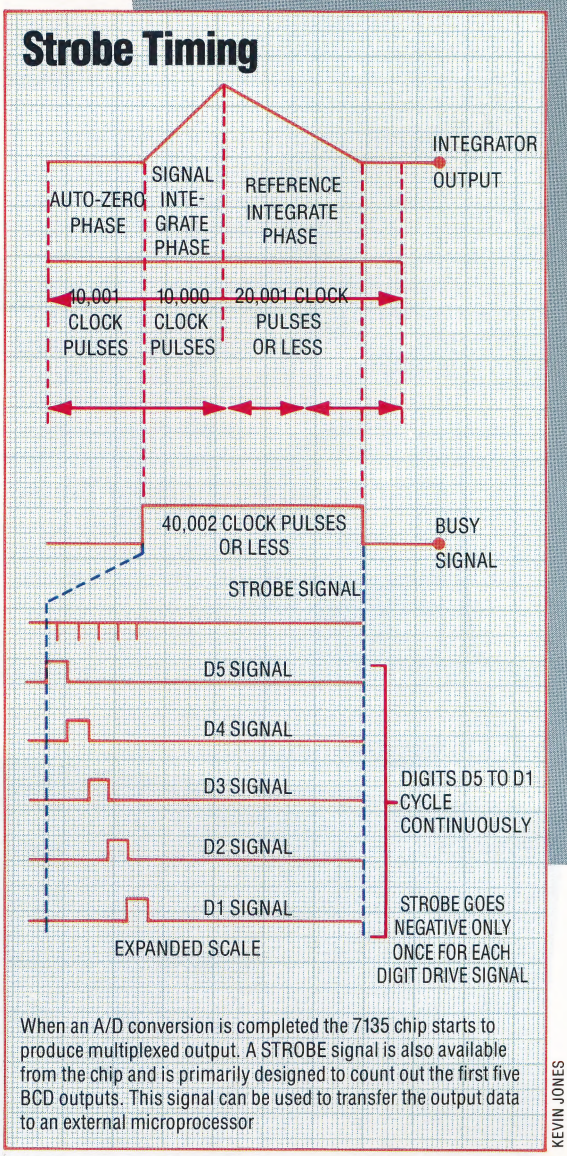
11. V+: A +5v power feed at a maximum current of 3mA is required to supply the chip.

12. D5: There are five digit enable outputs, D1-D5. (The other outputs are on pins 17, 18, 19 & 20.) They are used to enable the seven-segment LED displays one at a time so that they can be multiplexed. Each digit enable output is a positive going signal that stays high for 200 clock pulses. The five digits are scanned continuously, starting with D5, the most significant digit (MSD) down to D1, the least significant digit (LSD). As each digit drive output goes active, its voltage rises from near-ground level (about 0.25v) to near the positive rail voltage (about 4v). When this happens, the transistor to which it is connected is turned on and allows current to flow through segments of the associated digit (see circuit diagram on page 1516). The digit drive signals only *enable* each of the seven-segment LED displays, by connecting the common anode to the +5v rail. The individual segments of the display are switched on by the BCD signals (B1, B2, B4 and B8) as decoded by the seven-segment decoder/driver chip, the 7447. We have looked at logic circuits that convert a four-line BCD input to a seven-line output that lights the appropriate bars of a seven-segment display in our series on logic (see page 206).

13. B1 (LSB): There are four BCD (binary coded decimal) output lines: B1 on pin 13, B2 on pin 14, B4 on pin 15 and B8 (the MSB) on pin 16. Between them they provide a binary coded equivalent of the decimal digit to be displayed at any moment. The BCD code produced will be the one appropriate for the digit that is currently enabled. The chip's digit enable signals are synchronised so that the BCD value for any particular digit is present on the output lines when the corresponding enable signal is high.

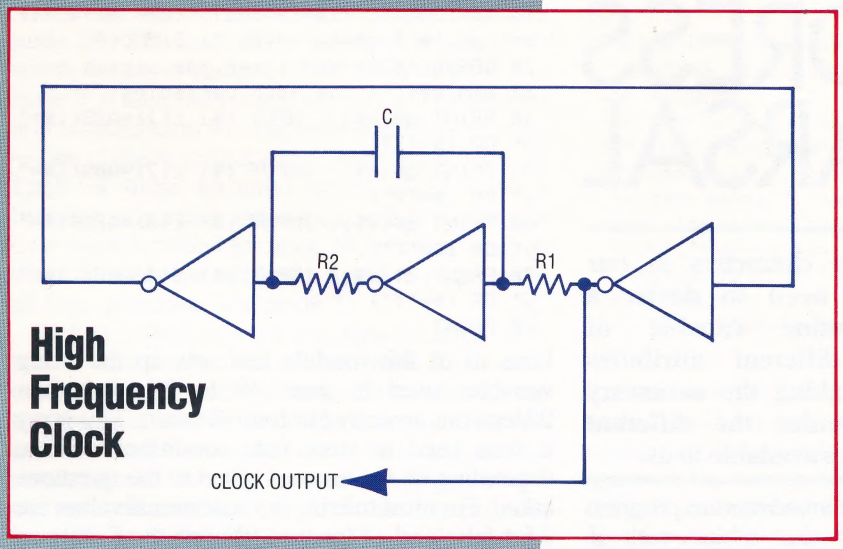
All five digits will appear to be on at the same time. In fact, they are multiplexed, flashing on in sequence one after the other. This gives a light output intensity apparently much lower than if the LEDs were illuminated continuously. However, it is not practical to power all five displays simultaneously. Because of their enormous power consumption, the LEDs would require nearly five times the energy of a circuit with just one LED on at a time.

More importantly, separate BCD outputs



When an A/D conversion is completed the 7135 chip starts to produce multiplexed output. A STROBE signal is also available from the chip and is primarily designed to count out the first five BCD outputs. This signal can be used to transfer the output data to an external microprocessor

KEVIN JONES



High Frequency Clock

Keeping Time

This alternative to the 555 clock chip uses just three TTL inverters, two resistors and a capacitor. The formula for calculating the desired frequency is:

$F = 2C(R1+R2)$

As the 7135 convertor chip will tolerate relatively high clocking rates, this type of clock circuit could be used to synchronise its actions if high frequency, low accuracy measurements were required

would be required for each digit — 20 pins instead of nine pins, and each display would require its own 7447 display driver chip — five chips instead of one. And the only advantage would be a brighter display. Multiplexing results in a large reduction in component and wiring costs.

21. BUSY: This is an important line if you want to interface the DVM with your micro. It is an 'active high' signal, which means that when it is high, the A/D converter is busy making calculations, and the digit and BCD outputs have no real meaning and should be ignored. It goes high at the beginning of the signal integrate phase, and stays high until the first clock pulse after the zero crossing point. This means that when the BUSY signal goes low, the signal output is latched into a stable state and can be validly read by, for example, an external microcomputer. We have discussed this principle in the previous instalment (see page 1516).

22. CLOCK IN: An external clock source is used to trigger the 7135 chip's internal operations. This clock signal should be no more than 1.2MHz. This maximum frequency provides an extremely fast update on the voltage being measured, but such a high frequency creates many problems concerned with component selection and accuracy. In any event, the 555 chip that we'll be using cannot generate such high frequencies — it won't be pushed above 100KHz.

Since a complete measurement cycle takes a maximum of 40,002 clock pulses, the frequency of the clock will determine how many measurements will be made each second. A 100KHz clock will make approximately 2.5 measurements per second. If you need faster measurements, clock frequencies of up to 1.2MHz are possible, but you will need an alternative clock oscillator circuit and very high quality auto-zero and reference capacitors. Accuracy is likely to suffer as well. It says much of the flexibility of the 7135 chip that it can work with an enormous range of frequencies, even down to 1Hz — although in this case you would need to wait 40,002 seconds for a reading!

23. POLARITY: This pin produces a signal that simply indicates the polarity of the voltage being measured. It is positive for all positive input voltages, even ones so slow that the display shows +00000. It is used to switch on the '+' sign on the leftmost $\frac{1}{2}$ digit LED display.

24. DIGITAL GROUND: There is little to cover here that has not been adequately dealt with under pin 3 — Analogue Ground (see page 1516)

25. RUN/HOLD: When this input pin is high the 7135 is in 'free run' mode, giving output readings every 40,002 clock pulses or less. If it is taken low (either by a HOLD signal from a computer, or by a front panel switch on the DVM), the current reading will be held indefinitely. After being held, a positive signal on this pin lasting at least 300 nanoseconds will start a new measurement cycle. This pin would be useful for a slow parallel computer interface if all five BCD outputs could not be processed quickly enough.

26. STROBE: There are five strobe pulses going negative — one for each valid digit — in each measurement cycle. They are timed to occur in the centre of each digit enable output pulse (see Timing Chart). The STROBE signal is provided to simplify interfacing to microprocessors and can be used to transfer the BCD data to external latch registers. The STROBE signal goes true (negative) for only half a clock pulse, and only once for each digit. The digit drivers and BCD outputs continue to cycle during the current measurement cycle (to enable a continuous display to appear), but a parallel computer interface would have to catch the STROBE pulses when they occurred — no more will be output until the next measurement cycle (see the Timing Chart).

27. OVER RANGE: This signal indicates that the input voltage being measured is too high (more than 20,000 counts). In a more sophisticated design it could be used as part of an auto-ranging circuit — a circuit that automatically sets the input sensitivity without the use of mechanical switches. In a simpler circuit such as ours, it could be used to illuminate an LED to show that the input attenuator should be set to a lower sensitivity (higher range).

28. UNDER RANGE: This signal goes high when the input voltage being measured is too low. Like the OVER RANGE signal, it is primarily intended for use in an auto-ranging circuit. Auto-ranging DVMs use the UNDER and OVER RANGE signals to switch the input attenuator to an input sensitivity appropriate for the signal being measured, but auto-ranging increases the complexity of the design and this signal will not be used in our circuit.

We have concluded our look at the design principles of DVMs and the signals available on the 7135 A/D converter. In the next instalment we shall begin construction of the DVM.

DRESS REHEARSAL

In order to bring the characters in our program to life, we need to devise a character-handler routine capable of processing all the different attributes concerned and then taking the necessary action. Here, we examine the different programming techniques available to us.

You can usually recognise an adventure program written in BASIC — it's choc-a-bloc with IF statements. One well-known book on BASIC adventure programming features a sample program with no less than four pages of IFs and THENs, which might lead you to conclude that this method of decision-making is inevitable when writing in BASIC.

The problem arises in adventures because of the large number of variables we need to process in order to decide what action to take. For example, each of the characters in the Dog and Bucket has 10 attributes, and our character-handler needs to be able to assess and update each of these while manipulating the 'person' concerned. During execution of the handler, we might, for example, have a routine that operates roughly as follows: 'IF character X is in the lounge AND character Y is in the lounge AND IF character Y is holding character X's drink AND character X is holding something, THEN character X throws object at character Y'.

If that and other similar routines are programmed into our game, it looks as if we are going to have lots of IF statements, whether we like it or not. This isn't very desirable, not least because of the amount of typing we'll have to do to enter the listing! What's needed, therefore, is a way of either drastically reducing the number of IFs, or — even better — doing without them altogether, since contrary to what you might imagine, there *are* viable alternatives to endless IF statements. Not only are these alternatives more compact, but they also tend to be faster in execution, so we win out on all counts.

FASTER ALTERNATIVES

In order to list the various possibilities open to us, let's look at a simple program that enables us to set up a number of conditions, test them and act accordingly. We'll then examine different ways of coding it, and decide on the best method to adopt for our character-handler. First enter the 'low level subroutines module' (lines 4000 to 4140) as printed in the previous instalment (and appropriately 'flavoured' if you own a BBC, Commodore or Spectrum computer), then type in these lines:

```
10 h$="human ":a$="animal ":m$="male ":f
$="female ":q$="Are you ": DIM c(4)
20 GOSUB 4050: REM clear the screen
30 REM set up the four variables
40 PRINT q$;h$;: INPUT i$: c(1)=ABS(i$="
y" OR i$="Y")
50 PRINT q$;a$;: INPUT i$: c(2)=ABS(i$="
y" OR i$="Y")
60 PRINT q$;f$;: INPUT i$: c(3)=ABS(i$="
y" OR i$="Y")
70 PRINT q$;m$;: INPUT i$: c(4)=ABS(i$="
y" OR i$="Y")
80 PRINT
```

Line 10 of this module first sets up the string variables used in lines 40 to 70, and then DIMensions an array c to four elements. This array is then used to store four conditional values, depending on the answers given to the questions asked. For most micros, the conditional values are 0 for false and -1 for true (though the Spectrum uses 1). The negative condition explains why we have included ABS in each assignment.

RUNning this module will set up the array c(4), ready for testing later. However, it's at this point that the problems arise, since with four different conditions there are a possible 16 different combinations. If you only wanted to test one instance — whether, for example, the user had entered "Y" to the first question and "N" to all the others — you could easily use an IF . . . THEN construct. However, if you wanted to test, say, 15 of the 16 possible combinations, you could end up with 15 lines like this:

```
90 REM Print messages
100 IF c(1)=1 AND c(2)=1 AND c(3)=1 AND
c(4)=1 THEN PRINT "You're a mixed up wer
ewolf!"
110 IF c(1)=1 AND c(2)=1 AND c(3)=1 AND
c(4)=0 THEN PRINT "You're a female werew
olf!"
```

. . . and so on. This is obviously rather long-winded, and there are other methods we can consider.

The first is to use the ON . . . GOTO construct (if your micro supports it — most do), but this also has its limitations since it requries a sequence of values and tends to be rather inflexible. As a general rule, however, it is a very much underused alternative to IF statements and on some machines (the Amstrad, for example) also has the advantage of executing more quickly.

The second method to explore is in fact the one we shall adopt for use in our character-handler. It involves rethinking the problem to a certain extent and will provide us with a powerful means of sorting different combinations of conditions as they arise in the Dog and Bucket. The answer lies in the implementation of 'decision trees' for our characters.

Let's go back to the four conditions we set up in our program and, instead of selecting the conditions we want to respond to by the use of IF statements, see how we might represent the decision process as a tree structure. The diagram shows how such a tree might appear.

You can see the first condition to be tested, at node 1, is 'Human?'. If the answer is yes (in other words the arrray variable c(1)=1) then we branch to node 3; if false, we branch to node 2. This process is continued until we reach the bottom of the tree — one of the 'terminal nodes', numbered 16 to 31. Each of these terminal nodes could have a message or routine associated with it. Examples have been included for node 16, which is reached when the user has entered a negative response to all four questions, and node 31.

Before trying to work out how to implement such a structure within our program, there are a couple of other points to note. First, you can see that the tree is divided into four 'levels', and that on each level all the conditions to be tested are the same. Level 1 tests the 'human' condition, level 2 the 'animal' condition, level 3 the 'female', and level 4, the 'male'. By design, the level numbers coincide with the elements of the array c(4), so c(1) holds the 'human' condition, and so on.

The next point to notice is the relationship between the different node numbers. Each choice node branches to two lower nodes, and the numbers of these may be determined by:

lower node number = (choice node number * 2) + condition value

where the condition value, held in the c array, is either 0 or 1.

Enter the third listing and RUN the program, whereupon you should find yourself greeted with an appropriate message, depending on how you answered the four questions.

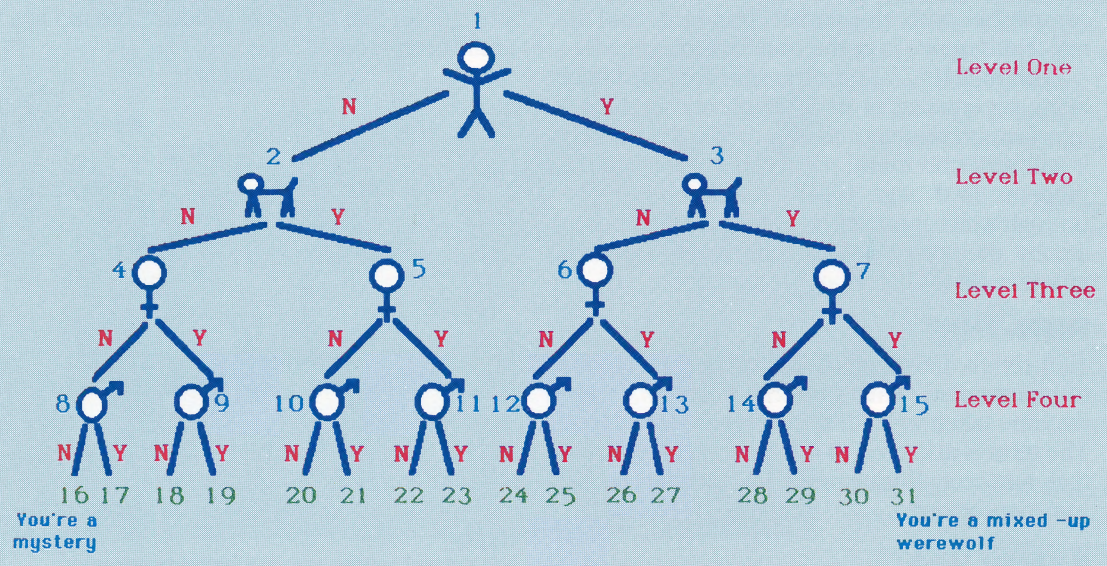
```
90 REM sort through the tree
100 n=1: REM start at node 1
110 FOR k=1 TO 4: n=(2*n)+c(k): NEXT k
120 RESTORE: REM set data pointer to sta
rt of messages
130 FOR x=1 TO n-15: READ z$: NEXT x: PR
INT "You're a ";z$: PRINT
140 PRINT "Hit a key to continue..."
150 GOSUB 4130: REM get a character
160 GOTO 20
170 REM data for messages
180 DATA "mystery","male mineral?!","fem
ale mineral?!","ridiculous time-waster",
"neuter animal","male animal","female an
imal","snail?  They can be bisexual, you
 know","human, sex unknown","man","woman
","mixed-up kid","neutered werewolf","ma
le werewolf"
190 DATA "female werewolf","mixed-up wer
ewolf"
```

Let's see how this listing works. We want to start at node 1, and keep track of where we are in the tree, so the first thing we do is initialise a variable n to represent the current node number. Line 110 then does all the business of moving down through the tree, level by level, ending at one of the terminal nodes. It does this by using the variable k to represent the level and applying the formula described above, node number = (2*node number) + c(level number), to decide which branch to follow down through the tree.

Once we've traversed all four levels and arrived at one of nodes 16 to 31, it's a simple matter to subtract 15 from the terminal node number to give us a number between 1 and 16 and then, in line 130, read through the DATA store and select an appropriate message for printing. Lines 140 to 160 simply wait for you to press a key before repeating the process.

There are other ways we could have solved the problems in our 'human/animal' program. However, the usefulness of tree structures becomes more apparent when we wish to apply rather more irregular rules to our condition-processing. We shall be looking at this in more detail in the next instalment.

Tree Of Knowledge



A 'decision tree' can be a useful way of determining program flow when a large number of conditions need to be taken into consideration. The tree shown is a simple binary one — more complex forms can be programmed in BASIC although this sort of data structure is most ideally suited to a language like LISP, LOGO, or PASCAL. Here the user enters values for four different conditions, which are then sorted by the tree to reach one of 16 different conclusions

CAROLINE CLAYTON

RS232C

Also known as a 'V24 port', the *RS232C* is a standard serial interface that was originated in the US by the Electronic Industries Association, and is now used throughout the world for serial transmissions. The trouble is, like all standards initially set for microcomputers, manufacturers have adapted them in order to meet their own needs. For example, there are a wide range of possible configurations for the output sockets, although these tend to be DIN types. However, for printers (one of the most common applications for the RS232C standard), the 25-way D connector is so widespread that it is now considered to be standard.

In order to use an RS232C system, all that is really required is a data-in line, a data-out line and a ground connection, although for synchronous two-way transmission, request to send (RTS) and clear to send (CTS) handshaking lines (see page 728) are also required.

Since the RS232C standard was established, there have been further variations produced, the most notable being the *RS422*. These are mostly compatible with the earlier standard, and the major difference is in the transmission speeds.

Although RS232-type transmissions are largely being superceded in many applications, such as interfaces to printers and disk drives, the standard is still very important, particularly with regard to modems (see page 1089). As telephone wires through which modems transmit their information are by their very nature serial lines, the serial interface clearly has a long life ahead of it.

RUN-TIME

Run-time is the period of time taken for a program to be executed. Thus run-time does not include the time taken to LOAD or assemble the program. The term is also used in reference to other events that may occur during program execution, such as 'run-time errors'.

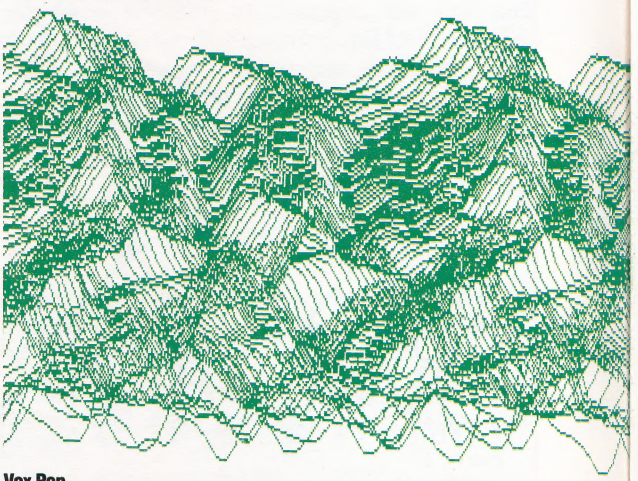
SAMPLING

This is a method of statistical analysis in which a *sample* of a total 'population' (which is the scope of items on which the analysis is to be carried out), is taken. The results of the analysis of that sample are extrapolated to encompass the whole population. Naturally, the larger and more representative the sample size in relation to the total population, the more accurate the results of the analysis.

When choosing a sample, there are two main methods of selection. The first is 'random sampling' in which the sample chosen to represent the total population is drawn completely by random selection techniques (see *random number*, page 1440). The second method of choosing samples is known as 'systematic sampling'. In this system, the members of the sample are chosen at regular intervals. For example, in a list of 100 items, choosing every fifth item is a systematic sampling.

Another form of sampling, which is becoming increasingly important in the field of electronic music, samples an analogue signal by analysing it for a small length of time. After a certain interval (either a fixed length of time or when the signal changes), another sample will be taken. It should be noted that the higher the frequency of the analogue signal the greater the number of samples that should be taken, otherwise the sample will cease to be an accurate reflection of the signal.

Because it is extremely difficult to copy 'normal' sounds, like the tone of a saxophone, simply by adjusting the oscillators and filters on a standard synthesiser, contemporary musicians using electronic equipment have increasingly turned to sampling methods. Using this approach, the sound of the saxophone will be input into a computer via a microphone and the note will be sampled. The computer then analyses the various frequencies and envelopes that make up the note. The machine can thus not only reproduce the exact sound that was played into the microphone, but can also alter it so that, for example, 'saxophone' scales can be played at a keyboard or the pitch changed so that the notes produced sound like an entire brass section.



Vox Pop
This picture of a print-out from the Fairlight music system shows the waveform obtained by 'sampling' a human voice. Once the sound has been digitally recorded and placed in memory, it is possible to alter the frequencies stored to produce a radically different sound

SAWTOOTH

So called because it resembles in shape the teeth of a saw, the *sawtooth wave* is a triangular-shaped wave that has an almost perpendicular rise time to the wave's maximum value, and a sloping, less dramatic, decay period back to the minimum (usually zero volts).

While it is impossible for digital devices to produce a 'pure' sawtooth wave, as they cannot produce the characteristic 'triangular' decay shape, it can be approximated by a number of small steps that can be built up to produce the correct shape. It is, however, very easy for analogue circuits, using capacitors, to generate these types of wave forms as the slow decay of charge is ideal.

HOT OFF THE PRESS

CHRIS STEVENS

In the previous instalment we outlined the way that optics and laser technology may be used in computers of the future. In fact, the Apple LaserWriter already makes use of this technology in producing the highest-quality print yet seen from any microcomputer.

Most printers built for microcomputers are either the dot matrix or daisy wheel variety. Although these are adequate for most applications, each has its own drawbacks. Dot matrix printers (see page 304), although fast and capable of producing an enormous range of different characters, produce a typeface that rarely stands up to close scrutiny. The average dot matrix printer, which has around 8 x 10 pins on its head, produces a rather ragged outline. On the other hand, daisy wheel printers (see page 364) can produce high quality print, but lack the speed and versatility of the dot matrix variety. For example, a daisy wheel has only a single fount (unless the user is willing to change the wheel), with a maximum speed of around 80 characters per second; whereas a dot matrix can produce text at 2–300 characters per second, although at that speed the letter quality is likely to suffer still further.

Laser printers are capable of outputting quality printed pages at phenomenal speeds. For example, the £200,000 IBM 3800 Model 3 laser printer, for use with their mainframe systems, can produce 215 pages of text per minute.

The Apple LaserWriter, which is a scaled-down version of the IBM-style laser printer, can produce high quality print at 300 characters per second. Furthermore, the user can choose from a number of different founts from a menu provided. It can also produce diagrams, forms and other types of artwork to a standard beyond the capabilities of conventional printers.

The unit is designed for use with the Apple Macintosh and connects to the computer via the same high-speed serial connection used for other types of printers.

First Impressions
Although the LaserWriter is intended for use with micros, it is totally unlike the dot matrix and daisy wheel printers used with most microcomputers. A more meaningful comparison is with fast laser printers for minis and mainframes costing tens of thousands of pounds. The LaserWriter is, of course, slower than mainframe equivalents, but the advanced technology is similar

The LaserWriter works in a radically different way from conventional units. Whereas dot matrix and daisy wheel printers are essentially developments of the typewriter system, with a moving head typing on to a roll of paper, the LaserWriter is more akin to a photocopying machine.

The printer measures 290 by 420 by 475mm and houses two drums, the first containing the photocopying toner while the other holds the laser equipment and the rotating cylinder used to print the design on to paper.

The printing mechanism, designed by Canon, works by taking the page to be printed and converting it into a bit image. The image is then transferred on to the cylinder, which is covered by millions of electrically sensitive dots. Before the transfer takes place, the dots are ionised so that they hold a positive electrical charge. In order to transfer the image, the laser scans across the cylinder, 'writing' the page on to its surface. Where the laser beam strikes the dots their positive charge

will be neutralised. The toner, which is also positively charged is then used to cover the surface of the cylinder. The toner will 'stick' where the dots have been neutralised, but avoid the rest as they hold the same positive charge and will therefore repel each other.

In order to transfer the toner on to the paper, the LaserWriter puts a negative charge on the paper as it passes into the machine. As the paper passes over the drum, the positively charged toner will be attracted to the negatively charged paper and will stick to it. In order to fix the toner permanently on to the page, the paper is then passed through a pair of rollers which are heated to a temperature of 200 degrees Centigrade. The toner is fused on to the paper which then passes out of the machine. Using this system, up to eight complete pages can be produced by the LaserWriter per minute. In use, the LaserWriter generates a large amount of heat, though this is not caused by the laser itself but is due to the heat produced by the rollers.

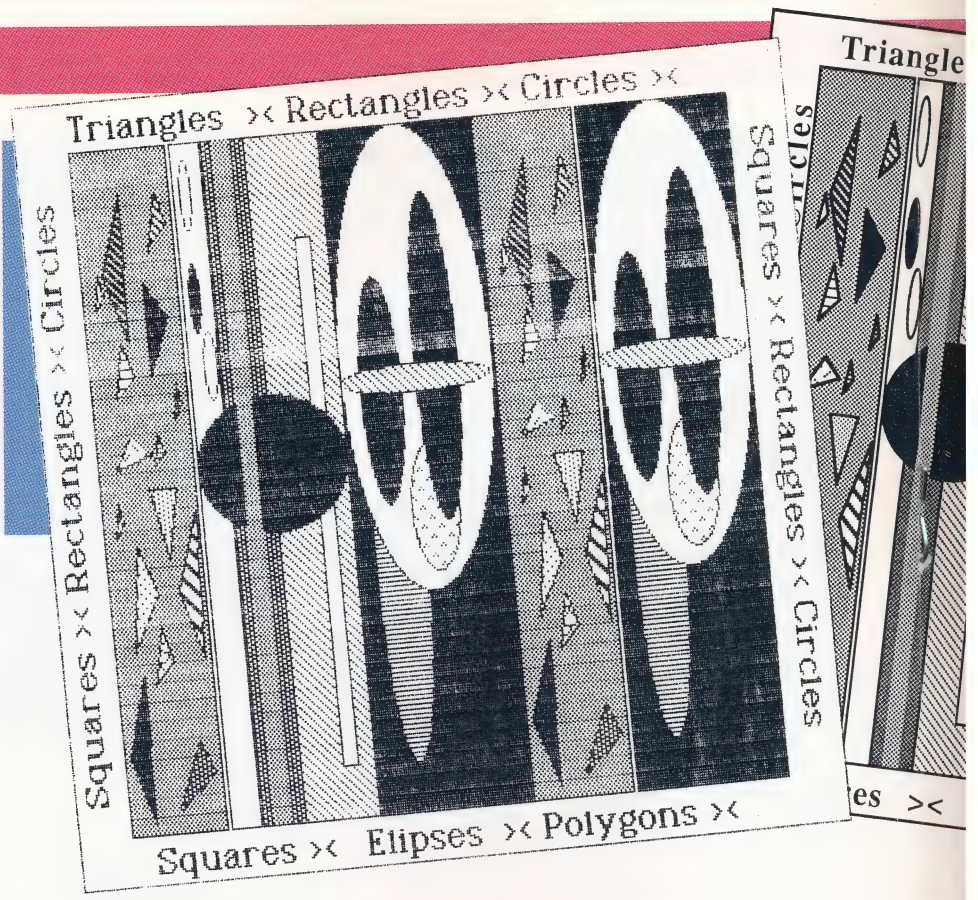
After around 3,000 pages have been produced, some parts of the laser system begin to wear out. These parts are fixed to the toner drum, and when the toner runs out, there is no method of refilling the drum, so the entire drum together with the worn-out parts has to be replaced. At around £100 for a replacement, this might seem like an expensive system, but Apple point out that at about three pence a copy, the LaserWriter compares well with most ordinary photocopiers.

PRINT QUALITY

The quality of the print produced approaches that of photo-typesetting (like that used to produce this page). This is achieved by the number of dots that Canon have managed to cram on to the cylinder surface — 300 dots per inch. For an A4–sized page this means that almost seven million dots are used in its production.

This system of laser printing, although fast and efficient, is by no means unique. What distinguishes the LaserWriter from similar devices on the market, is the way that Apple has incorporated its own technology into the machinery, making it the first 'intelligent' printer. We have already seen that the LaserWriter places an image of the page into its memory before printing begins. Bearing in mind the very high resolution of the resulting print, a considerable amount of memory needs to be available. The LaserWriter incorporates an on-board computer dedicated solely to producing the printed page.

Based around the Motorola 68000 processor, running at 12 MHz, the same one used in the Macintosh itself, the LaserWriter has 1.5 Mbytes of RAM and 500 Kbytes of ROM. The ROMs hold full character sets for up to 13 different typefaces. However, even 2 Mbytes cannot hold full bit-mapped images of all of these, and all the application programs required by the LaserWriter. Therefore, in order to conserve memory, the printer's ROM merely holds the outline of each character with a special routine which 'fills in' the characters when they are printed. This system not only saves space but by using the same 'template', the outline can be expanded to accommodate any

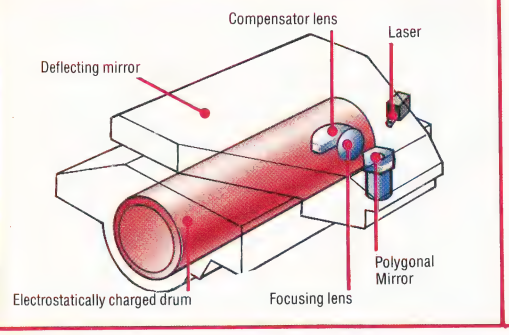


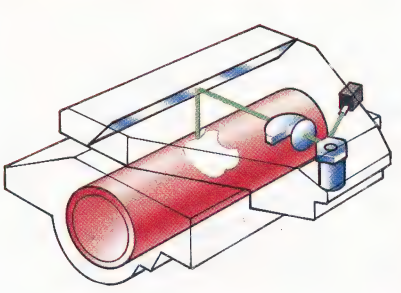
A Highly Charged Procedure
This series of illustrations shows how the LaserWriter fixes an image on paper. Although the technology used is extremely sophisticated, the principles are fairly simple, based around Xeroxing techniques which have been in use for many years

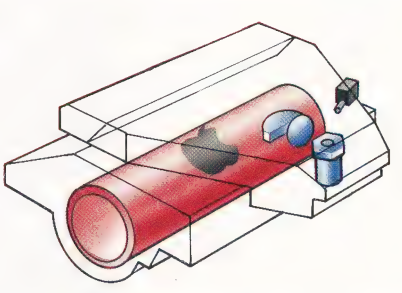
The drum is given an electrostatic positive charge

The laser beam is then fired and deflected on to the drum where it draws the image by neutralising the positive ions

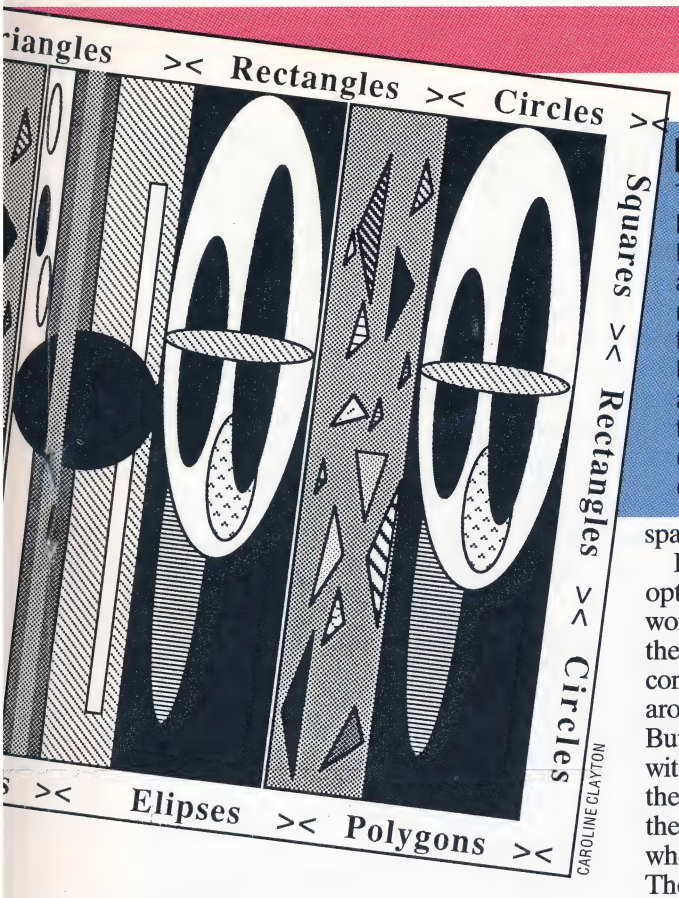
As the drum's surface passes the toner container, the positively-charged toner will be attracted to the 'neutralised' areas and collect on them







KEVIN JONES



Drawing Comparisons

This design has been reproduced on both the Macintosh Imagewriter dot matrix printer and the LaserWriter. Apart from the problems that might arise from varying ink density on the dot matrix ribbon, the resolution of the LaserWriter is far higher than the Imagewriter. A close examination of the Imagewriter design reveals unintended lines appearing. This is where the characters formed by the print head overlap. This problem does not arise on the LaserWriter because the page image is fixed on a single drum

type size required.

Also held in ROM, is a special language developed for use on printers known as PostScript. This is an interactive programming language which allows the programmer to take the fount outlines held in ROM — and other facilities such as lines — and manipulate them to produce many styles and designs. The commands which make up the language can be used to create routines to produce circles and so on, which, in turn, can be fitted together to produce complex diagrams. In operation, the language closely resembles FORTH in its heavy use of a stack to pass data to and from procedures and its ability to define other procedures. PostScript works differently from other graphics programming languages such as LOGO. Instead of telling the printer to draw a series of lines in absolute terms, PostScript uses a series of points (such as those which make up the fount outlines) which will be spaced according to a series of proportions.

However, PostScript is more than a desirable optional extra, it is fundamental to the efficient working of the LaserWriter. In order to transfer the entire one Mbyte of information required for a complete page, it would take the Macintosh around 35 seconds to transmit the information. But the programs on the systems disk provided with the LaserWriter convert this information into the Macintosh's Quickdraw format. The data is then sent down the serial line to the LaserWriter where it is converted to PostScript commands. These, in turn, are interpreted by the LaserWriter's computer to reproduce the page as a bit-mapped image within the computer's memory. This bit-map is then used by the laser to 'draw' the page on to the elecrosensitive drum. By reducing the information in this way, as little as eight Kbytes of information is sufficient, dramatically reducing the transmission speed and the amount of processing.

The LaserWriter is undoubtedly a major step forward in printer technology — not only in the advanced laser mechanism used to produce fast and high quality print, but also in the on-board computer provided and its use of the PostScript programming language. Of course, the price tag puts it well beyond the reach of most Macintosh users. However, advanced technology has a habit of filtering down to lower-priced machines and in a few years we can expect much of the leading edge technology used on the LaserWriter to become available on cheaper equipment.

APPLE LASERWRITER

PRICE
£8,050 inc VAT

DIMENSIONS
290×420×475mm

INTERFACES
RS422 and RS232C serial interfaces

PRINT RESOLUTION
300 dots per inch

CPU
Motorola 68000 processor running at 12MHz

MEMORY
1.5 Mbytes of RAM and 500 Kbytes of ROM

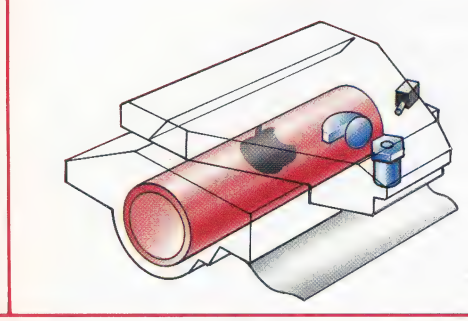
STRENGTHS
The LaserWriter produces professional quality letters and diagrams at a speed far greater than most other printers designed for micros

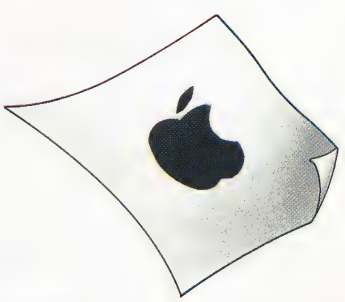
WEAKNESSES
The maximum paper size is a little over standard A4, which severely restricts the extent to which artwork and designs can be produced. Furthermore, the price tag will restrict the machine to all but the wealthiest micro owners

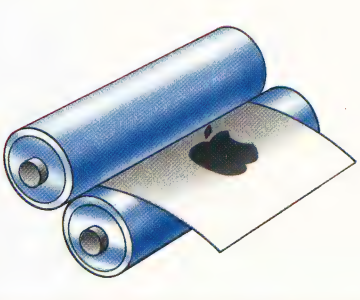
Negatively-charged paper is then fed into the drum . . .

. . . where it attracts the positively-charged toner and forms an image on the page

Finally, the paper holding the toner is passed between heated rollers which 'fix' the toner on the page







KEVIN JONES

OBLIQUE STRATEGIES

We complete the Go programs for Amstrad, Commodore 64 and Sinclair Spectrum owners by adding the final move evaluation routines. These routines are based on the methods of group and pattern analysis first developed for the BBC Micro on page 1512.

One of the reasons that the game of Go has proved so difficult to program is that it is almost impossible to simulate the workings of the human mind using present-day technology. To determine moves, the human player can analyse intuitively the group and line patterns formed by the stones. As current micros can only work sequentially — that is to say, one step at a time — analysing group shapes presents a problem. However, we can include routines that can analyse connections between adjacent groups, allowing the computer to connect and defend two of its groups, or attack possible connections between opposing groups. In this instalment, we give routines that perform these functions on the Amstrad, Commodore 64 and Spectrum micros.

Module Five

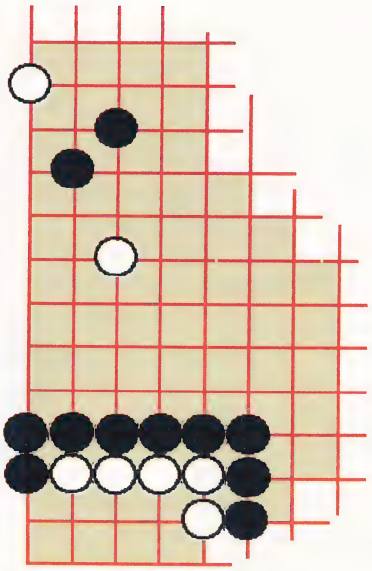
Commodore 64:

```
330 GOSUB 790
780 :
790 REM READ-PATTERNS ROUTINE
810 PAT=BOARD+556
830 FOR L=0 TO 71
840 READ P%:POKE PAT+L,(256+P%) AND 255
850 NEXT
860 DATA 32,17,16,2,-15,1
870 DATA -32,-17,-16,-2,15,-1
880 DATA 33,16,17,31,16,15
890 DATA -33,-16,-17,-31,-16,-15
900 DATA -14,1,-15,18,1,17
910 DATA 14,-1,15,-18,-1,-17
920 DATA 48,33,32,48,17,16
930 DATA -48,-33,-32,-48,-17,-16
940 DATA 3,-14,2,3,-15,1
950 DATA -3,14,-2,-3,15,-1
960 DATA 64,33,32,4,-14,2
970 DATA -64,-33,-32,-4,14,-2
980 RETURN
990 :
1000 REM*********************************
2580 IF LOCAT%=0 THEN GOSUB 2900:T$="DEF
"
2590 IF LOCAT%=0 THEN GOSUB 3070:T$="ATT
"
2600 IF LOCAT%=0 THEN GOSUB 3230:T$="SAT
"
2610 IF LOCAT%=0 THEN GOSUB 3380:T$="SCN
"
2890 :
2900 REM DEFEND-CONNECTION
2910 HI=-9999
2920 BV%=BLACK%:GOSUB 4420
2930 FOR A=1 TO 255:DCS%=PEEK(BOARD+A):I
F DCS%<>BLACK% GOTO 3010
2940 FOR P=PAT TO PAT+70 STEP 3
2950 B=PEEK(BOARD+((A+PEEK(P))AND 255)):
C=PEEK(BOARD+((A+PEEK(P+1))AND 255))
2960 IF B<>BLACK% OR C<>WHITE% GOTO 3000
2970 D=(A+PEEK(P+2))AND 255:SCR=RND(0)+P
EEK(WEIGHT+D)
2980 IF (D AND 240)=0 OR (D AND 15)=0 OR
 SCR<=HI GOTO 3000
2990 LP%=D:LC%=BLACK%:GOSUB 3890:IF LL%=
0 AND CLIB%>2 THEN HI=SCR:LOCAT%=D
3000 NEXT
3010 NEXT
3020 BV%=0:GOSUB 4420
3030 RETURN
3040 :
3050 REM*********************************
3060 :
3070 REM ATTACK-CONNECTION ROUTINE
3080 HI=-9999
3090 BV%=WHITE%:GOSUB 4420
3100 FOR A=1 TO 255:ACS%=PEEK(BOARD+A):I
F ACS%<>WHITE% GOTO 3180
3110 FOR P=PAT TO PAT+70 STEP 3
3120 B=PEEK(BOARD+((A+PEEK(P))AND 255)):
C=PEEK(BOARD+((A+PEEK(P+1))AND 255))
3130 IF B<>WHITE% OR C<>BLACK% GOTO 3170
3140 D=(A+PEEK(P+2))AND 255:SCR=RND(0)+P
EEK(WEIGHT+D)
3150 IF (D AND 240)=0 OR (D AND 15)=0 OR
 SCR<=HI GOTO 3170
3160 LP%=D:LC%=BLACK%:GOSUB 3890:IF LL%=
0 AND CLIB%>2 THEN HI=SCR:LOCAT%=D
3170 NEXT
3180 NEXT:BV%=0:GOSUB 4420
3190 RETURN
3200 :
3210 REM*********************************
3220 :
3230 REM START-ATTACK ROUTINE
3240 HI=-9999
3250 FOR A=1 TO 255:SAS%=PEEK(BOARD+A):I
F SAS%<>WHITE% GOTO 3330
3260 FOR P=PAT TO PAT+70 STEP 3
3270 B=PEEK(BOARD+((A+PEEK(P))AND 255)):
D=PEEK(BOARD+((A+PEEK(P+2))AND 255))
3280 IF B<>WHITE% OR D<>0 GOTO 3320
3290 C=(A+PEEK(P+1))AND 255:SCR=RND(0)+P
EEK(WEIGHT+C)
3300 IF (C AND 240)=0 OR (C AND 15)=0 OR
 SCR<=HI GOTO 3320
3310 LP%=C:LC%=BLACK%:GOSUB 3890:IF LL%=
0 AND CLIB%>2 THEN HI=SCR:LOCAT%=C
3320 NEXT
3330 NEXT
3340 RETURN
3350 :
3360 REM*********************************
3370 :
3380 REM START-CONNECTION ROUTINE
3390 HI=-9999
3400 FOR A=1 TO 255:SCS%=PEEK(BOARD+A):I
F SCS%<>BLACK% GOTO 3470
3410 FOR P=PAT TO PAT+70 STEP 3
3420 C=PEEK(BOARD+((A+PEEK(P+2))AND 255)
):IF C>0 GOTO 3460
3430 B=(A+PEEK(P))AND 255:SCR=RND(0)+PEE
K(WEIGHT+B)
3440 IF (B AND 240)=0 OR (B AND 15)=0 OR
 SCR<=HI GOTO 3460
3450 LP%=B:LC%=BLACK%:GOSUB 3890:IF LL%=
0 AND CLIB%>2 THEN HI=SCR:LOCAT%=B
3460 NEXT
3470 NEXT
3480 RETURN
3490 :
3500 REM*********************************
4410 :
4420 REM BOUNDARY ROUTINE
4430 FOR BX=0 TO 15
4450 BY=16*BX
4460 POKE BOARD+BX,BV%
4470 POKE BOARD+BY,BV%
4480 NEXT
4490 RETURN
4500 :
4510 REM*********************************
4520 REM**********END OF PROGRAM*********
```

Amstrad CPC 464/664:

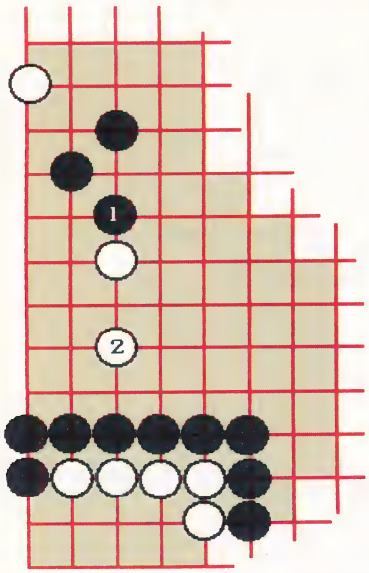
```
280 GOSUB 790:REM read patterns
780 :
790 REM read patterns routine
810 pat=board+&300
820 RESTORE 860
830 FOR i%=0 TO 72
840 READ p%:POKE (pat+i%),(256+p%)AND 25
5
850 NEXT i%
860 DATA 32,17,16,2,-15,1
870 DATA -32,-17,-16,-2,15,-1
880 DATA 33,16,17,31,16,15
890 DATA -33,-16,-17,-31,-16,-1
900 DATA -14,1,-15,18,1,17
910 DATA 14,-1,15,18,-1,-17
920 DATA 48,33,32,48,17,16
930 DATA -48,-33,-32,-48,-17,-16
940 DATA 3,-14,2,3,-15,1
950 DATA -3,14,-2,-3,15,-1
960 DATA 64,33,32,4,-14,2
970 DATA -64,-33,-32,-4,14,-2
980 RETURN
990 :
1000 REM ************************
2580 IF location%=0 THEN GOSUB 2900:T$="
DEF"
2590 IF location%=0 THEN GOSUB 3070:T$="
ATT"
2600 IF location%=0 THEN GOSUB 3230:T$="
SAT"
2610 IF location%=0 THEN GOSUB 3380:T$="
SCN"
2900 REM defend connection routine
2910 hi=-9999
2920 bv%=black%:GOSUB 4420:REM boundary
2930 FOR a%=1 TO 255:dcs%=PEEK(board+a%)
:IF dcs%<>black% THEN 3010
2940 FOR p%=pat TO pat+70 STEP 3
2950 b%=PEEK(board+((a%+PEEK(p%))AND 255
)):c%=PEEK(board+((a%+PEEK(p%+1))AND 255
))
2960 IF b%<>black% OR c%<>white% THEN 30
00
2970 d%=(a%+PEEK(p%+2)) AND 255:score=RN
D(1)+PEEK(weight+d%)
2980 IF (d% AND 240)=0 OR (d% AND 15)=0
OR score<=hi THEN 3000
2990 lp%=d%:LET lc%=black%:GOSUB 3890:IF
 li%=0 AND clib%>2 THEN hi=score:locatio
n%=d%
3000 NEXT p%
3010 NEXT a%
3020 bv%=0:GOSUB 4420:REM boundary
3030 RETURN
3040 :
3050 REM ************************
3060 :
3070 REM attack connection routine
3080 hi=-9999
3090 bv%=white%:GOSUB 4420:REM boundary
3100 FOR a%=1 TO 255:acs%=PEEK (board+a%
):IF cs%<>white% THEN 3180
3110 FOR p%=pat TO pat+70 STEP 3
3120 b%=PEEK(board+((a%+PEEK(p%))AND 255
)):c%=PEEK(board+((a%+PEEK(p%+1))AND 255
))
3130 ifb%<>white% OR c%<>black% THEN 317
0
3140 d%=(a%+PEEK(p%+2))AND 255:scr%=RND(
1)+PEEK(weight+d%)
3150 IF (d% AND 240)=0 OR (d% AND 15)=0
OR score<=hi THEN 3170
3160 lp%=d%:lc%=black%:GOSUB 3890:IF li%
=0 AND clib%>2 THEN hi=score:loaction%=d
%
3170 NEXT p%
3180 NEXT a%
3190 RETURN
3200 :
3210 REM ************************
3220 :
3230 REM start attack routine
3240 hi=-9999
3250 FOR a%=1 TO 255:sas%=PEEK(board+a%)
:IF sas%<>white% THEN 3330
3260 FOR p%=pat TO pat+70 STEP 3
3270 b%=PEEK(board+((a%+PEEK(p%))AND 255
)):d%=PEEK(board+((a%+PEEK(p%+2))AND 255
))
3280 IF b%<>white% OR d%<>0 THEN 3320
3290 c%=(a%+PEEK(p%+1)AND 255):score=RND
(1)+PEEK(weight+c%)
3300 IF (c% AND 240)=0 OR (c% AND 15)=0
OR score<=hi THEN 3320
3310 lp%=c%:lc%=black%:GOSUB 3890:IF li%
=0 AND clib%>2 THEN hi=score:location=c%
3320 NEXT p%
3330 NEXT a%
3340 RETURN
3350 :
3360 REM ************************
3370 :
3380 REM start connection routine
3390 hi=-9999
3400 FOR a%=1 TO 255:scs%=PEEK(board+a%)
:IF scs%<>black% THEN 3470
3410 FOR p%=pat TO pat+70 STEP 3
3420 c%=PEEK(board+((a%+PEEK(p%+2))AND 2
55)):IF c%>0 THEN 3460
3430 b%=(a%+PEEK(p%))AND 255:score=RND(1
)+PEEK(weight+b%)
3440 IF (b% AND 240)=0 OR (b% AND 15)=0
OR score<=hi THEN 3460
3450 lp%=b%:lc%=black%:GOSUB 3890:IF li%
=0 AND clib%>2 THEN hi=score:location%=b
%
3460 NEXT p%
3470 NEXT a%
3480 RETURN
3490 :
3500 REM ************************
4410 :
4420 REM boundary routine
4440 FOR x%=0 TO 15
4450 y%=16*x%
4460 POKE (board+x%),bv%
4470 POKE (board+y%),bv%
4480 NEXT x%
4490 RETURN
4500 :
4510 REM ************************
4520 REM ***** end of program *******
```

Thickness

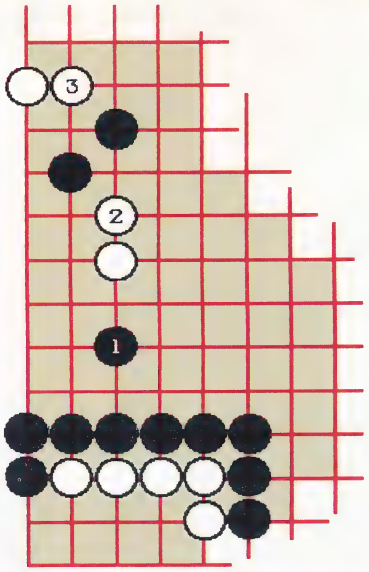


In The Thick
Solidly formed, outward-looking groups, such as the black group shown here, are said to have 'thickness'. As there are no weak spots in a thick group, attacks upon them are likely to fail. An important strategic axiom in Go is: 'stay away from thickness'

CAROLINE CLAYTON



Thick As Thieves
It is generally a good strategy to use your own thick groups to assist in an attack on enemy stones. In the situation shown here, Black 1 plays above the single white stone, thus driving White towards the thick black group at the bottom of the board. White may try to extend out from the attacked stone by playing stone 2, but this is unlikely to be successful in the long run, as the move is closer to the area of black thickness



Foiled Again
If Black plays on the lower side of the white stone, the attack is likely to fail as White can simultaneously defend itself and attack the two black stones nearby by playing stone 2. Under threat of a white move into position 3, black is forced onto the defensive and the attack crumbles

Sinclair Spectrum:

```
 280 GO SUB 790
 790 REM read-patterns routine
 810 LET pat=board+556
 820 RESTORE 860
 830 FOR l=0 TO 71
 840 READ p: POKE pat+l,p
 850 NEXT l
 860 DATA 32,17,16,2,-15,1
 870 DATA -32,-17,-16,-2,15,-1
 880 DATA 33,16,17,31,16,15
 890 DATA -33,-16,-17,-31,-16,-1
5
 900 DATA -14,1,-15,18,1,17
 910 DATA 14,-1,15,-18,-1,-17
 920 DATA 48,33,32,48,17,16
 930 DATA -48,-33,-32,-48,-17,-1
6
 940 DATA 3,-14,2,3,-15,1
 950 DATA -3,14,-2,-3,15,-1
 960 DATA 64,33,32,4,-14,2
 970 DATA -64,-33,-32,-4,14,-2
 980 RETURN
1000 REM ************************
2580 IF location=0 THEN  GO SUB
2900: LET t$="DEF"
2590 IF location=0 THEN  GO SUB
3070: LET t$="ATT"
2600 IF location=0 THEN  GO SUB
3230: LET t$="SAT"
2610 IF location=0 THEN  GO SUB
3380: LET t$="SCN"
2900 REM defend-connection routi
ne
2910 LET hi=-9999
2920 LET bv=black: GO SUB 4420
2930 FOR a=1 TO 255: LET dcs=PEE
K (board+a): IF dcs<>black THEN
 GO TO 3010
2940 FOR p=pat TO pat+70 STEP 3
2950 LET b=a+PEEK p: LET c=a+PEE
K (p+1)
2952 IF b>255 THEN  LET b=b-256:
 GO TO 2952
2954 IF c>255 THEN  LET c=c-256:
 GO TO 2954
2956 LET b=PEEK (board+b): LET c
=PEEK (board+c)
2960 IF b<>black OR c<>white THE
N  GO TO 3000
2970 LET d=a+PEEK (p+2)
2972 IF d>255 THEN  LET d=d-256:
 GO TO 2972
2974 LET score=RND+PEEK (weight+
d)
2980 IF INT (d/16)=0 OR d-16*INT
 (d/16)=0 OR score<=hi THEN  GO
TO 3000
2990 LET lp=d: LET lc=black: GO
SUB 3890: IF ll=0 AND clib>2 THE
N  LET hi=score: LET location=d
3000 NEXT p
3010 NEXT a
3020 LET bv=0: GO SUB 4420
3030 RETURN
3070 REM attack-connection routi
ne
3080 LET hi=-9999
3090 LET bv=white: GO SUB 4420
3100 FOR a=1 TO 255: LET acs=PEE
K (board+a): IF acs<>white THEN
 GO TO 3180
3110 FOR p=pat TO pat+70 STEP 3
3120 LET b=a+PEEK p: LET c=a+PEE
K (p+1)
3122 IF b>255 THEN  LET b=b-256:
 GO TO 3122
3124 IF c>255 THEN  LET c=c-256:
 GO TO 3124
3126 LET b=PEEK (board+b): LET c
=PEEK (board+c)
3130 IF b<>white OR c<>black THE
N  GO TO 3170
3140 LET d=a+PEEK (p+2)
3142 IF d>255 THEN  LET d=d-256:
 GO TO 3142
3144 LET score=RND+PEEK (weight+
d)
3150 IF INT (d/16)=0 OR d-16*INT
 (d/16)=0 OR score<=hi THEN  GO
TO 3170
3160 LET lp=d: LET lc=black: GO
SUB 3890: IF ll=0 AND clib>2 THE
N  LET hi=score: LET location=d
3170 NEXT p
3180 NEXT a: LET bv=0: GO SUB 44
20
3190 RETURN
3230 REM start-attack routine
3240 LET hi=-9999
3250 FOR a=1 TO 255: LET sas=PEE
K (board+a): IF sas<>white THEN
 GO TO 3330
3260 FOR p=pat TO pat+70 STEP 3
3270 LET b=a+PEEK p: LET d=a+PEE
K (p+2)
3272 IF b>255 THEN  LET b=b-256:
 GO TO 3272
3274 IF d>255 THEN  LET d=d-256:
 GO TO 3274
3276 LET b=PEEK (board+b): LET d
=PEEK (board+d)
3280 IF b<>white OR d<>0 THEN  G
O TO 3320
3290 LET c=a+PEEK (p+1)
3292 IF c>255 THEN  LET c=c-256:
 GO TO 3292
3294 LET score=RND+PEEK (weight+
c)
3300 IF INT (c/16)=0 OR c-16*INT
 (c/16)=0 OR score<=hi THEN  GO
TO 3320
3310 LET lp=c: LET lc=black: GO
SUB 3890: IF ll=0 AND clib>2 THE
N  LET hi=score: LET location=c
3320 NEXT p
3330 NEXT a
3340 RETURN
3380 REM start-connection routin
e
3390 LET hi=-9999
3400 FOR a=1 TO 255: LET scs=PEE
K (board+a): IF scs<>black THEN
 GO TO 3470
3410 FOR p=pat TO pat+70 STEP 3
3420 LET c=a+PEEK (p+2)
3422 IF c>255 THEN  LET c=c-256:
 GO TO 3422
3424 LET c=PEEK (board+c): IF c>
0 THEN  GO TO 3460
3430 LET b=a+PEEK p
3432 IF b>255 THEN  LET b=b-256:
 GO TO 3432
3434 LET score=RND+PEEK (weight+
b)
3440 IF INT (b/16)=0 OR b-16*INT
 (b/16)=0 OR score<=hi THEN  GO
TO 3460
3450 LET lp=b: LET lc=black: GO
SUB 3890: IF ll=0 AND clib>2 THE
N  LET hi=score: LET location=b
3460 NEXT p
3470 NEXT a
3480 RETURN
4420 REM boundary routine
4440 FOR x=0 TO 15
4450 LET y=16*x
4460 POKE board+x,bv
4470 POKE board+y,bv
4480 NEXT x
4490 RETURN
4500:
4510 REM ************************
4520 REM ****END OF PROGRAM*****
```

RATIONAL SAVINGS

For reasons of speed and efficiency, FORTH's colon definitions are 'semi-compiled'. We examine FORTH's methods of compilation, and discuss their implications for the re-defining of existing words and program execution.

An important feature of FORTH is that its colon definitions are semi-compiled. Although they are not converted into machine code (as in full compilation), they are processed to take up less space and be executed more quickly. As a result, FORTH programs are very efficient.

There are various methods of semi-compiling. The figFORTH method is by far the most common, to the extent that other methods are considered unorthodox. Suppose you define a word:

```
: SQUARE ( n — n*n) DUP * ;
```

First of all, a *header* is put into the dictionary. This contains (the actual order of entry may differ between versions of FORTH) the name SQUARE and the memory address of the header for the previously defined word. Thus all the headers are linked to each other and, starting with the newest, the computer can search back through the entire dictionary until it finds the word it's looking for. These two parts of the header are called the *name field* and the *link field.*

After the header comes the definition. First there is a code that shows what kind of definition it is: there are different codes for colon definitions (as in this case), variables, constants, the structures set up by CREATE, words defined in machine code for the system and for any other kinds of words. This part is called the *code field.*

Code Fields

Code fields are rather clever. The code number is actually the memory address of the machine code that's run whenever the word is executed. For any colon definition, this machine code remembers where the FORTH interpreter must return to when it has finished this particular definition. This is done by putting the return address on another stack, the *return stack* — and redirecting the interpreter to the present definition.

There are different machine code routines for constants and variables (which have a single number in their parameter fields). The so-called primitive words, like + and *, are written in machine code. This is stored in the parameter field and the code field contains its address

Next comes the main part of the colon definition. This is a list of the addresses of the words it uses (actually the addresses of the code fields of those words, also known as their *compilation addresses*). This main part is called the *parameter field.*

This is what you end up with:

name field:	"SQUARE"
link field:	address of header of previous word (2 bytes)
code field:	code for a colon definition (2 bytes)
parameter field:	compilation address of DUP (2 bytes)
	compilation address of * (2 bytes)
	compilation address of ; (2 bytes)

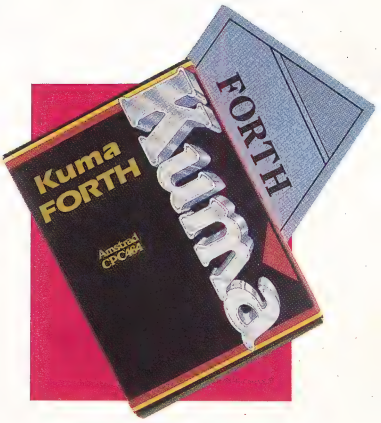
You can now begin to see how colon definitions are compiled. First, the name and link fields are set up. This is the same for any new word definition. Next, : puts in the special code for colon definitions, and it also puts FORTH into a special compiling state. From now on, when you enter a word it's not executed in the normal way, but its compilation address is enclosed in the dictionary. The OK message is also suppressed to let you know this is happening.

This continues until ;, so obviously ; must be special. Not only is it enclosed in the dictionary, but it also has the immediate effect of telling FORTH to revert from compiling to its normal state. Words like this, which must be executed straight away as part of the compilation process, are called *immediate* words. They aren't enclosed in the dictionary unless they see to it themselves. Actually, what ; encloses in the dictionary is not its own compilation address, but that of an anonymous shadow, the run-time action for ;.

When SQUARE is executed (for instance if you enter 4 SQUARE .) FORTH looks it up in the dictionary and finds its definition. This then tells it to execute in turn DUP and * and then stop, but it doesn't need to look these up in the dictionary because it already has the addresses of their definitions. Colon definitions can therefore be executed quite fast.

This describes the simplest colon definitions. Numbers and program structures both introduce complications.

Numbers clearly don't have compilation addresses, so they are compiled in a composite form: first, the compilation address of a *literal handler* (another anonymous shadow), and then the number itself. When executed from within a



Amstrad FORTH
The Amstrad CPC 464/664 join other home micros in their ability to run FORTH, thanks to this package from Kuma Software. The Kuma version of FORTH is an implementation of figFORTH, with floating point, string, and colour graphics facilities

colon definition, the literal handler's effect is to tell the system 'the next two bytes you see aren't a compilation address to be executed, but a number to be put on the stack'. ." has a similar composite of handler followed by the string itself.

The next complication is with programming structures like IF . . . ELSE . . . THEN. If you were programming in machine code — or in the more primitive BASICS — you'd have to do these with jumps, something like:

```
1000 If TopOf Stack = 0 THEN GOTO 1100
1010 REM do this lot if the condition was true
      :
      :
1090 GOTO 1200
1100 REM do this lot if the condition was false
      :
      :
1200 REM carry on here in either case
```

Compiled FORTH is quite like this. There are two FORTH words, BRANCH (unconditional jump) and ?BRANCH (remove top of stack and jump if it's zero), but you can't use them in the normal course of events because they have to be used as composite forms (like a compiled number): the compilation address followed by the address of the place to jump to. It is the job of IF, ELSE, THEN etc., which are immediate words, to work out these jump addresses and compile the composite forms. For instance, IF encloses the compilation address of ?BRANCH and space for the jump address, but it can't fill the jump address in because it doesn't yet know where ELSE is. Instead, it leaves the address of this space on the stack so that ELSE can fill in the jump address. Actually, IF also leaves a check number on the stack. Each kind of structure has its own check number, so if ELSE doesn't find IF then it knows that something is wrong: either there never was an IF, or something in between has messed up the stack. This stops you writing things like:

```
IF . . . BEGIN . . . ELSE . . . UNTIL . . . THEN
```

Let's look at some examples.

1. Suppose that your version of FORTH distinguishes between upper and lower case, and that all standard words are entered in the dictionary in upper case. This would mean that loop would not be recognised, whereas LOOP would be. If you wanted to be able to enter lower-case and upper-case words, you might start off :

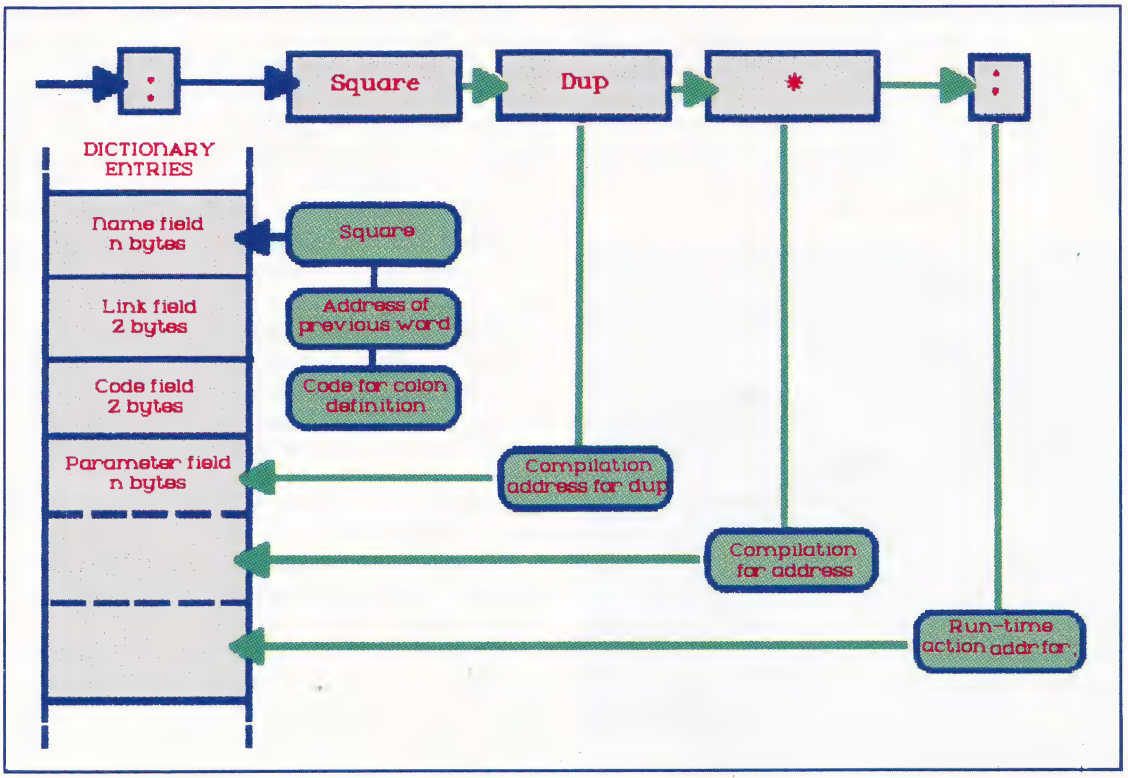
```
: drop DROP ;
: roll  ROLL ;
: variable VARIABLE (yes, this will work);
```

You run into trouble with :

```
: loop LOOP (this won't work) ;
```

```
: loop [COMPILE] LOOP; IMMEDIATE
```

You need to keep a clear head to see what this does, as there are three separate occasions to bear in mind. First you must consider what happens when loop is defined. Although LOOP is immediate, [COMPILE] overrides this so that LOOP's compilation address goes into the definition of loop. The second occasion, compile-time, is when loop is used in defining another word. Because it is immediate, it is executed straight away, and its effect is then to execute LOOP. This is the time to think about most, because it's when loop does its



Storing The Word
FORTH uses a method of 'semi-compilation' to aid program execution. Once a colon definition is entered, different fields are set up within the dictionary, starting with the name, link and code fields. FORTH then enters a 'compilation mode' (indicated in red), whereupon a word is not entered directly in the dictionary, but instead has its 'compilation address' entered in the parameter field of the definition. Once ; is encountered, FORTH reverts to 'execution mode' (indicated in black)

Some Relevant Words

[and] allows you to switch temporarily to the execute state while you are still in the middle of a colon definition.

LITERAL, an immediate word, removes the top of the stack and compiles it into a colon definition in the usual way for a number. You'd typically use it with [and] to evaluate an expression at compile time and compile the result. For example

```
[ 9 16 * ] LITERAL
```

has the same effect in a colon definition as 144.

[COMPILE] is an immediate word and says that the next word should be enclosed rather than executed, even if it is immediate.

COMPILE is not immediate. When executed, it encloses the following word into the dictionary.

BRANCH and ?BRANCH are used as in the example when you are constructing jumps. >MARK and >RESOLVE are used to fill in the jump address for a forward jump: >MARK is used at the jump and >RESOLVE at its destination. <MARK and <RESOLVE are similar for backward jumps, with <MARK at the destination and <RESOLVE at the jump.

IMMEDIATE makes the previous word immediate.

stuff. The third occasion, run-time, is when that other word is executed, and then whatever LOOP has compiled into its definition comes into effect.

2. Here is a pair of words ?DO and ?LOOP. They are just like DO and LOOP, but if the start is already as big as the limit then the loop never gets executed.

```
?DO is equivalent to     OVER OVER > IF DO
?LOOP is equivalent to   LOOP ELSE DROP DROP
                         THEN
```

However, if you try to define:

```
: ?DO ( no good ) OVER OVER > IF DO ;
```

you have the same problem as with loop. The correct definition is:

```
: ?DO
    COMPILE OVER
    COMPILE OVER
    COMPILE >
    [COMPILE] IF
    [COMPILE] DO
; IMMEDIATE

: ?LOOP
    [COMPILE] LOOP
    [COMPILE] ELSE
    COMPILE DROP
    COMPILE DROP
    [COMPILE] THEN
; IMMEDIATE
```

3. Case structures provide a choice of actions depending on the value of something. They are not in the standard, but there is a commonly used one that you can define yourself. Its form is:

```
value CASE
1st possibility OF ... ENDOF
2nd possibility OF ... ENDOF
3rd possibility OF ... ENDOF
   ...
What to do if it doesn't match any possibility
ENDCASE
```

For instance, the value might be the ASCII code for a key pressed and you can specify different actions for any different characters it might be.

If you look at the jumps needed to do this, you'll see that each OF needs a conditional jump to its ENDOF, and each ENDOF needs a jump to ENDCASE. ENDCASE might need to fill in, or *resolve*, several jump addresses. There are no words in standard FORTH that do this, so you're forced to work out the jumps yourself for CASE, which you do with >MARK and >RESOLVE. >MARK stacks the address of the space for a jump address, as IF does, and >RESOLVE takes it off the stack and fills in the space.

```
: CASE ( run: value  — value
       ( compile:    — 0 )
  0    ( number of OFs — none so far )
; IMMEDIATE

: OF   ( run: value, possibility —          [if match] )
       (      value, possibility — value   [no match] )
       ( compile: — mark for jump to ENDOF )
   COMPILE OVER
   COMPILE =
   COMPILE ?BRANCH >MARK
   COMPILE DROP ( drop value if match )
; IMMEDIATE

: ENDOF ( run: — )
   ( compile: OF  count, OF's mark —
   (              mark for jump to ENDCASE,
                  OF count + 1)
   COMPILE BRANCH >MARK
   SWAP >RESOLVE  ( resolve jump from OF )
   SWAP 1+        ( increment OF count )
; IMMEDIATE
: ENDCASE ( run: value —
            ( compile: marks from ENDOFs,
            OF
            count — )
   COMPILE DROP
   0 ?DO
      >RESOLVE
   ?LOOP
; IMMEDIATE
```

Although this is complicated, the fact that you can do it at all shows how extendable FORTH is. In practice, you'd also use check numbers to ensure a CASE structure is put together properly.

FORTH-79 hasn't got BRANCH, >MARK, >RESOLVE and so on, and although figFORTH has similar words, you have to delve deeper in order to use them.

AMSTRAD ON LOCATION

As an introduction to our series on the Amstrad CPC 464 and 664 operating system, we take a look at the way in which ROM and RAM memory can be switched in and out. We also consider the various ways of gaining access to the Amstrad's firmware routines using a special area of memory known as a jumpblock.

There is very little of the latest silicon technology inside either the Amstrad CPC 464 or 664. However, the features that computer engineers usually implement in the hardware are compensated in the Amstrad machines by a more comprehensive operating system. In this way, a minimal amount of hardware was required to provide the facilities available on the Amstrad machines, resulting in a less expensive micro.

The CPC 464 uses a Z80A CPU, accompanied by 64 Kbytes of RAM, 32 Kbytes of ROM, an 8255 PIA (peripheral interface adaptor, see page 1289), an AY-3-8912 sound generator, a 6845 CRTC (cathode-ray tube controller) and a custom gate array (ULA — uncommitted logic array).

The operating system, known as the 'firmware', occupies the lower half of the 32 Kbyte ROM. The remainder of the ROM contains Amstrad's version of BASIC, called Locomotive BASIC. The firmware is divided into sub-sections, each of which deals with a particular aspect of the system. These provide a handy framework on which to base our discussion of the Amstrad OS in this series of articles. For the moment, however, let's take a general overview of the system.

The System Memory Map diagram shows that both halves of the ROM appear as two logically separate 16 Kbyte ROMs, which can be switched in and out independently using the ULA. Notice that the single 32 Kbyte ROM overlays the first and last 16 Kbytes of RAM, the middle 32 Kbytes of RAM always being free for reading or writing to. The ULA also provides the facility to switch the upper ROM between any of up to 252 sideways expansion ROMs. The use of expansion ROMs will be dealt with in subsequent instalments.

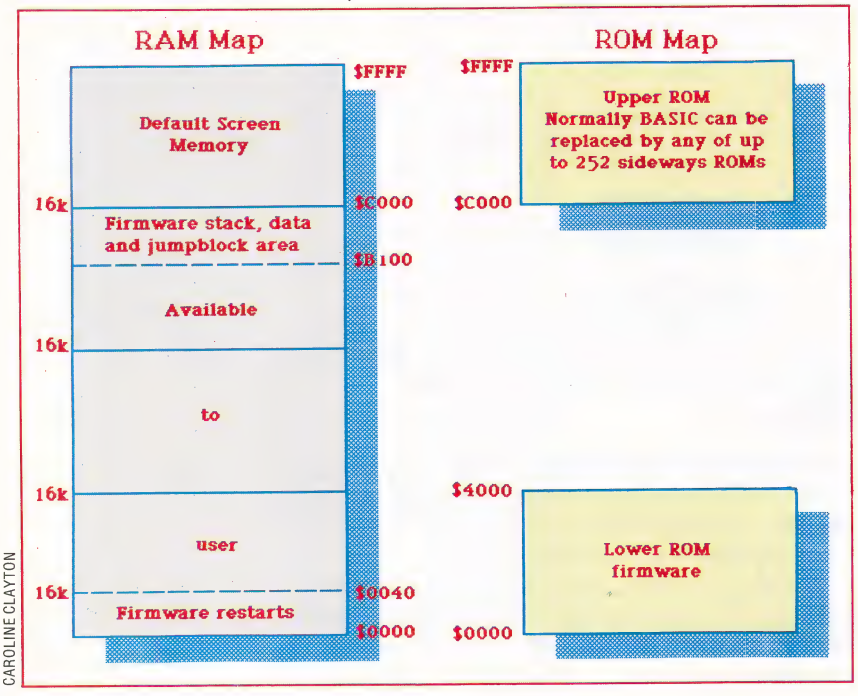
The screen is controlled by the 6845 chip and requires 16 Kbytes of RAM, which is usually provided by the memory under the top ROM. However, any of the four 16 Kbyte blocks may be chosen to hold screen data — thus allowing multiple screens to be switched in and out at lightning speeds.

When the top ROM is switched 'in' — for instance, when BASIC is under control — anything read from the top 16 Kbytes of memory is arranged by the ULA to return locations within the ROM. Anything written to this 16 Kbyte block passes through the ROM to the underlying RAM, which is usually occupied by the screen. To read from the RAM requires the top ROM to be switched 'out', and the firmware provides a variety of ways to do this.

A similar type of memory access applies to the lower ROM containing the firmware. But what happens if the lower ROM is switched 'out'? How can the operating system still run if its program has disappeared? This problem is overcome if the lower ROM is not directly accessed by programs, but instead all calls to the operating system are made via a *jumpblock*. This is an area of memory that has a fixed location, and which contains a series of entries that simply perform a jump (or similar action) to various routines within the firmware. The use of a jumpblock allows the actual addresses of the firmware routines to be changed, without affecting any existing programs. You should always call routines in the firmware via the jumpblock.

The jumpblock resides in the system RAM in the third of the 16 Kbyte blocks, and so cannot be switched out in the same way as the ROM. Provision is made to ensure that this area is not overwritten by the user. However, certain situations arise where it is desirable to alter the jumpblock in order to remove or replace part of the operating system. This is known as *jumpblock patching* and is discussed in detail later.

ROM Over RAM
Our Amstrad memory map shows clearly the manner in which the single 32K ROM chip 'pages out' the upper and lower 16K of RAM. The firmware can be used to switch in and out both pages of RAM, as well as check for the presence of up to 252 sideways ROMs. Screen data can be held in any of the four 16K blocks of RAM



CAROLINE CLAYTON

HOW THE JUMPBLOCK WORKS

All the entry points to the operating system have a corresponding entry in the jumpblock. When the user wishes to access the operating system, a call is made to the appropriate jumpblock entry. The jumpblock entry then issues an RST command, which in turn calls the firmware. Each entry is three bytes long; on power-up each contains an RST instruction followed by an operating system entry address.

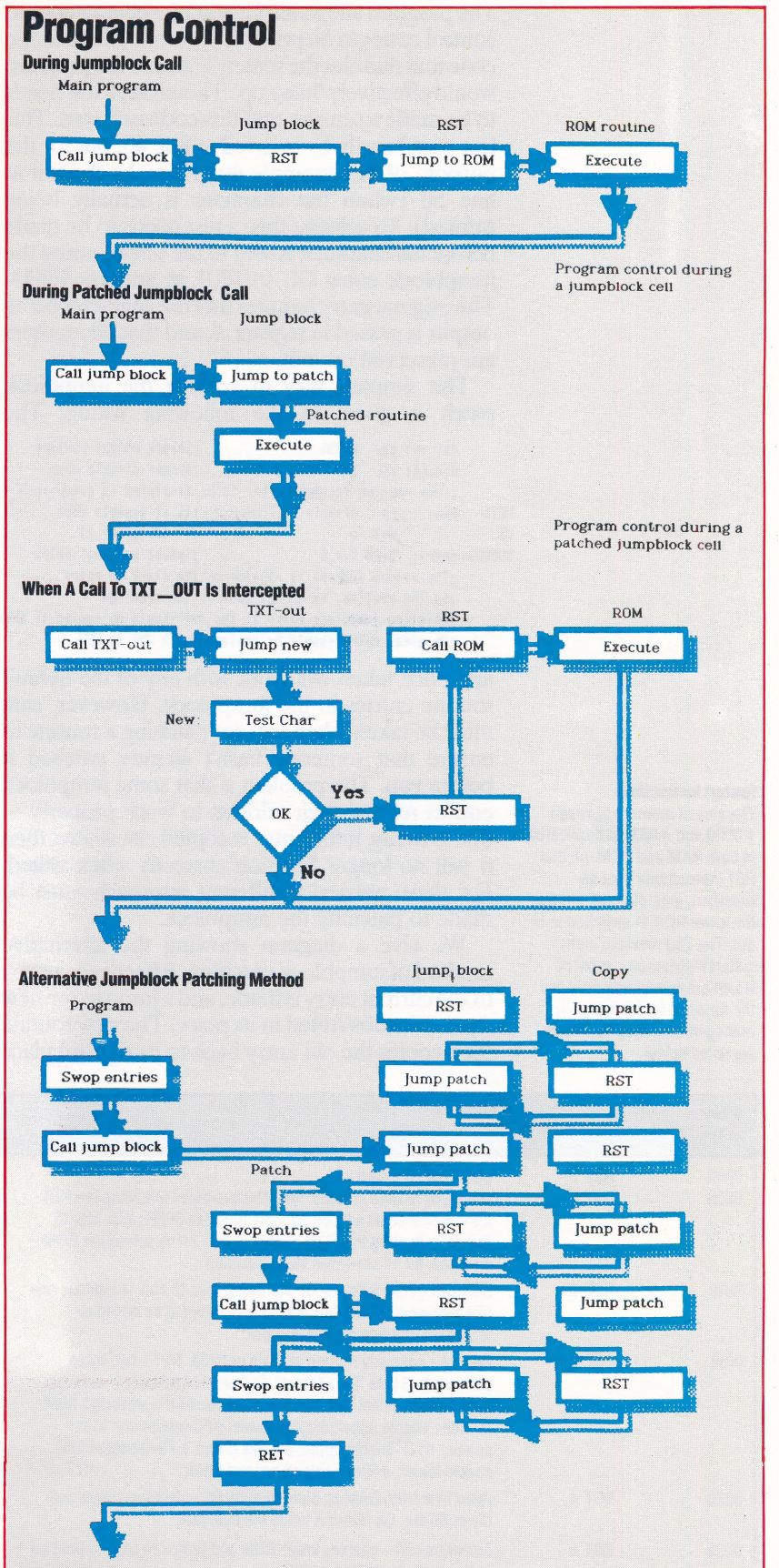
The lower ROM could be switched off when it is not needed, and for this reason a copy of the restart routines is maintained in the RAM adjacent to the ROM. These restart routines in turn switch on the firmware ROM before it is subsequently called. When the operating system routine returns, the ROMs are reset to their state on entry. This solves the problem of having to decide what ROMs to switch before calling the operating system. Most of the restart instructions are used to provide an extended instruction set; the extra commands are primarily to control ROM switching. The table overleaf lists the restart instructions and their uses.

The jumpblock system allows substitute routines to replace the existing operating system entries. For example, if you wanted to write your own screen-handling routines, these can be 'patched-in' to replace the existing routines by altering the jumpblock entry to point to a new routine. We give two diagrams that show the difference between the flow of control when a command is passed to the operating system normally, and the same chain when it has been intercepted by the user. This method of jumpblock patching is, in fact, the method used to switch between tape and disk commands on both the CPC 464 and 664.

As an example of jumpblock patching, suppose you have written a routine called SEND SCREEN, which sends text to the screen, and you wish this to be used instead of the existing operating system routine. The entry in the jumpblock for printing characters on the screen is called TXT WR CHAR and is addressed at 0BB5DH. The patch can be achieved by substituting a jump to the new routine in the jumpblock.

The procedure adopted above works well when an entirely new routine has been written and the original entry is no longer required. A problem starts to arise when either the original entry is still used, or a call to another entry in the jumpblock is required from within the new routine. Consider an example BASIC program that simply takes character codes and sends them to the screen:

```
10 MODE 1                    clear screen and homes cursor
20 WHILE -1                  set up an infinite loop
30 INPUT "Character          get the character code
   Code";C;
40 IF C<0 OR C>255           test for valid code
   then ?: GOTO 30
50 ?CHR$(C);                 send it out
60 WEND                      loop back
```



Hop, Skip and Jump
The jumpblock enables the user to make full and uncomplicated use of the firmware, and also to redirect system calls to user-defined routines. The latter process involves 'patching' the jumpblock and the diagrams here give a visual breakdown of the different steps involved, depending on which method is used

CAROLINE CLAYTON

The program allows all the graphics characters and control codes to be printed. However, there is one code that disables the screen — and so the program would effectively 'hang up'. Therefore, a test needs to be made to ensure that this code isn't sent. This test could easily be inserted at line 40, but for the purpose of this example we will make the test at line 50 (when the character is actually being printed). To achieve this, a test needs to be made before the character is sent to the screen, using the jumpblock entry TXT OUTPUT at address $BB5A. The original entry requires that the character being output is passed in register A and that all registers are preserved on exit.

The simplest way to achieve the jumpblock patch is given in the following listing. The

```
      txt_ou: equ  #BB5A              ;jblock output routine
      disabl: equ  21                 ;screen disable char
      ; New routine follows - char to be displayed is given in A
FE15  new:    cp   disable            ;is it disable VDU?
C8            ret  z                  ;if so ignore it
000000 entry: defb 0,0,0              ;insert original entry
      ;The return address is already on the stack on entry
      ;to the routine, so no RET instruction required
      ;Existing jumpblock entry at txt_out should be copied to ENTRY
      ;Original entry should be replaced with JMP to NEW
```

approach taken will work with any of the default routine entries in the jumpblock. However, care must be taken when you are patching a routine to ensure that someone hasn't already patched it before you. The problem is that some jumpblock entries rely on their address to work properly — that is to say, if the entry is copied elsewhere, then it will no longer function correctly when called. For these entries, a different approach must be made to patching the jumpblock.

We give a diagram showing the alternative method of jumpblock patching. Effectively a copy of the current entry is made, and a jump to the new routine is substituted in its place. The new routine then copies the old entry back to its rightful place in the jumpblock and calls it. Before it returns, the jump to the new routine is once again placed in the jumpblock.

Our second machine code listing is a program that takes our example and patches it in the way we have just described. This approach appears at first sight to be more long-winded, but it is guaranteed to work whatever the original contents of the jumpblock.

```
      txt_ou: equ  #BB5D              ;jblock output routine
      disabl: equ  21                 ;screen disable char
FE15  new:    cp   disable            ;routine entry point
C8            ret  z                  ;ignore disable
      ;now switch old entry into jumpblock and call it
CD2529        call copy               ;switch entry
CD5DBB        call txt_out            ;send character
CD2529        call copy               ;copy the entry back
C9            ret
      ;
F5    copy:   push af                 ;must preserve all regs
C5            push bc
D5            push de
E5            push hl
0603          ld   b,3                ;no of bytes in entry
215DBB        ld   hl,txt_out         ;old entry
113F29        ld   de,txt_cpy         ;copy area
4E    cloop:  ld   c,(hl)             ;read current entry
1A            ld   a,(de)             ;and its replacement
77            ld   (hl),a             ;substitute it
79            ld   a,c
12            ld   (de),a             ;and save the old one
23            inc  hl                 ;point to next byte
13            inc  de
10F7          djnz cloop              ;do all three
E1            pop  hl
D1            pop  de
C1            pop  bc
F1            pop  af
C9            ret
      ; new jumpblock entry is store below until required
C3    txt_cp: defb #c3                ;code for jmp
1829          defw new                ;to substitute routine
```

It is important to realise that any substitute routine must appear to the firmware as though it is the default routine (that is, all parameters passed to and returned from the routine must have the same definition as those for the routine to be replaced). In the example this is simply achieved by preserving the returned registers while the copy is taking place.

It is worthwhile mentioning here the reason for placing the new routines below HIMEM. The system RAM allocation was described earlier, and showed that the large area of RAM free in the bottom three 16-Kbyte blocks is available for BASIC use. The lower limit is in fact fixed (although later in the series we will show how to reserve memory below the BASIC operating area), and the upper limit is set by the value of HIMEM. Effectively, a BASIC program 'grows' from the lower limit up towards HIMEM.

To reserve space for a machine code routine, we must ensure that BASIC is not allowed to access the same area and, from the description above, it is clear that this must be achieved by lowering HIMEM. For this process to be as economical as possible with the memory space left to BASIC, HIMEM is lowered only by the length of the routine that we wish to use (plus its data area) and the routine loaded into the recently liberated RAM.

Restart Instructions

The area of memory between #0000 and #003F is duplicated in both RAM and ROM, so that RST instructions may be executed, regardless of whether the lower ROM is switched in or out. The Z80 provides eight restart instructions, most of which are extensively used by the Amstrad for memory management. Their functions are listed below

Address in hex		Function
0000	RST 0	Reset (as on power-up)
0008	RST 1	Jumps to routine in lower 16K. The two bytes following the RST are considered as a 14 bit address (0000 to 3FFF), with bits 15 and 14 indicating the ROM enable state – bit 15 set=upper ROM disabled; bit 14 set=lower ROM disabled
0010	RST 2	Side call – RST followed by two bytes; bits 15 and 14 indicate one of 4 sideways ROMs; bits 0 to 13 give offset to be added to #C000 to produce routine address
0018	RST 3	Far call – calls any routine in ROM or RAM. RST + two bytes pointing to 3 byte 'far address'. Far address bytes 0/1=address; byte 2=ROM select byte as follows: #00–#FB = sideways ROM number, enable upper, disable lower; #FC=upper and lower enable; #FD=enable upper, disable lower; #FE=disable upper, enable lower; #FF=disable upper and lower
0020	RST 4	Read byte from RAM as pointed to by HL – disables upper and lower ROMs. On return A holds the byte read
0028	RST 5	Firmware call – enables lower ROM and jumps to routine pointed to by two bytes following RST
0030	RST 6	User restart. Bytes #30 to #37 left unused
0038	RST 7	Interrupt entry. Best left alone. If you wish to make use of mode 1 interrupts you should consider using the firmware 'event' facilities which will be covered later in the series

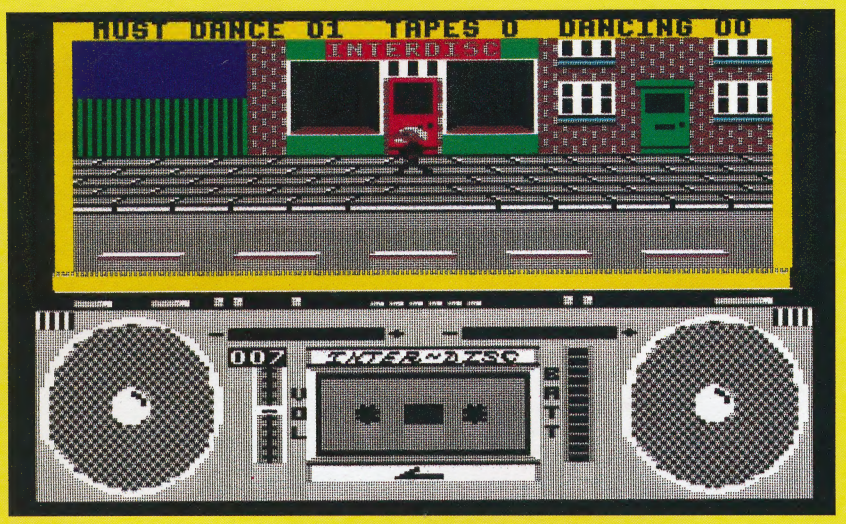
When a package claims to be the 'First Rock Musical written for computer' with 'the funkiest music yet on the CBM 64' it has to be a well-written program to withstand the disappointments that often accompany high expectations. Caroline Clayton sat in on a performance of Ghetto Blaster (Virgin Games, £7.95) and found it to be a witty, visual and tuneful treat.

A BLASTED GOOD TIME

Like all good games, the plot of Ghetto Blaster is relatively simple. Rockin' Rodney has just been employed as a messenger for a record company called Interdisc, and it is his job to deliver 10 demo tapes to the company's head office on Funky Street (just off Sunny Goodge Street) before the end of the day. He carries with him his ghetto blaster ('a prototype of the ultimate in powerful playback' says the inlay), on which he can play the demo tapes as he traverses the town. If you get him to aim the ghetto blaster at passing pedestrians and press the joystick, it will have a 'dance entrancing' effect on them. So far so good. But there are problems.

Rodney must perform his duties before the 'rev' counter on his ghetto blaster reaches 999, remembering all the time to watch out that the machine's batteries don't run low. He must avoid being mugged by the 'gangsters of the groove' (who want to snatch the demos), stay clear of the Psycho Killer and not come to the attention of the long arm of the law. Furthermore, his blaster can be damaged if he runs into the 'tone deaf walker', and he'll need to waste valuable time going to a repair shop to get it fixed. And on top of all that he has to locate the 10 demo tapes, which are hidden all over Funky Town (they are to found behind the pulsating doorways of quite ordinary houses). A simple plot, certainly, but add these complicating factors and you have a game with the potential to become very addictive.

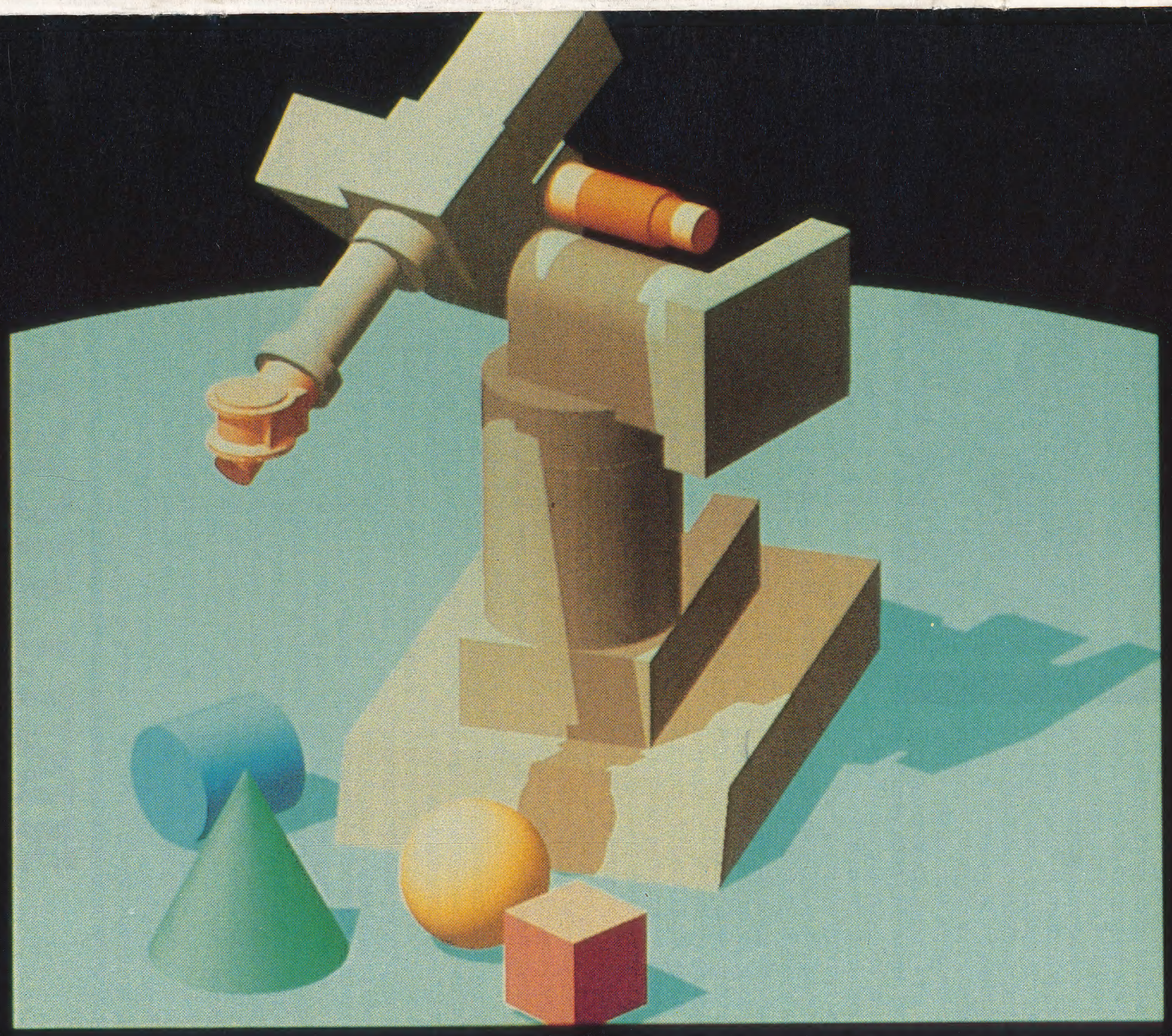
The screen display is split in two: the top half consists of a view of the street our Rodney is currently grooving down (the inlay claims this is in '3-D', which is overstating the case a little). However, the detail in the street scenes is quite thoughtful. Strawberry Fields and Blueberry Hill, as their names suggest, appear to be in a quite select part of town. Desolation Row and Stoney End, on the other hand, look depressingly dilapidated. The bottom half of the screen shows an enlarged ghetto blaster, which you must keep an eye on during the game as it indicates the time left (rev counter) and the battery power level. The volume indicator is also useful: to get the most playing time our of your batteries, of course, you should keep the volume level fairly low. This means you won't need to make so many visits to the electrical supplies shop to get fresh batteries.



The sound itself makes the most of the Commodore's funky musical potential. The first demo tape you pick up (teasingly entitled 'Relapse—Last Year's Tee Shirt Mix') sets the standard for the other tunes. No wonder that passers-by are so easily seduced into gyrating to the pulsating disco beat. But if you get bored grooving along the streets listening to your music, you can always go shopping (although you could be accused of being a 'material girl'), nip into one of the houses (unless you are told that 'I hear you knocking, but you can't come in') or take a 'trip' through Itchycoo Park. If you're lost, say a little prayer that you might run into Jumping Jack Flash (a guy with a good sense of direction). If you really get into dire straits, you can always go into 'pause mode' by pressing the Space bar.

Ghetto Blaster is basically a standard 'shoot-em-up' arcade game with blasts of tunes substituted for bullets, and this development is to be applauded. Like a good pop tune, it's a game that grows on you and sticks in your mind after you've played it. It is well-crafted and has a sense of humour (all too often missing from most packages).

The game can be appreciated on many levels, although as it's likely to be played by teenagers, its allusions to the psychedelic 'sixties may go a little unappreciated. Similarly, only David Essex fans may take delight in a street called Hold Me Close, but Ghetto Blaster is surely destined to become a mega-hit among those unable to appreciate its full subtleties.



© 1983 BAILEY, CHARLESWORTH, PLUNKETT,
HENDERSON—PURDUE CADLAB